Engenharia de Confiabilidade de Sites (SRE)

Aprimore tarefas operacionais com observabilidade, automação e gerenciamento de incidentes

Casa do Código | alura

MARCELO COSTA

Sumário

- ISBN
- Prefácio
- Parte 1: Fundamentos
  - 1 O que é SRE?
  - 2 Competências de SRE
  - 3 O ciclo de vida do desenvolvimento de software
- Parte 2: Infraestrutura
  - 4 Arquiteturas de software confiáveis
  - 5 Plataformas de runtime
- Parte 3: Observabilidade
  - 6 Métricas
  - 7 Logs
  - 8 Traces
- Parte 4: Conceitos de confiabilidade
  - 9 Objetivos de Nível de Serviço (SLOs)
  - 10 Boas práticas para alertas
- Parte 5: Gerenciamento de incidentes
  - 11 Ferramentas de SRE
  - 12 O ciclo de vida de um incidente
  - 13 Outras práticas para melhorias de confiabilidade
  - 14 Bibliografia

ISBN

Impresso: 978-85-5519-396-5

Digital: 978-85-5519-395-8

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Costa, Marcelo
Engenharia de confiabilidade de sites (SRE) : aprimore tarefas operacionais com observabilidade, automação e gerenciamento de incidentes / Marcelo Costa. -- 1. ed. -- São Paulo : AOVS Sistemas de Informática, 2025.

Bibliografia.
ISBN 978-85-5519-396-5

1. Confiabilidade (Engenharia) 2. Engenharia 3. Sites da Web - Avaliação 4. Sites da Web - Desenvolvimento 5. Sites da Web - Projetos I. Título.

25-252050 CDD-620.07

Índices para catálogo sistemático:

1. Engenharia de confiabilidade : Estudo e ensino 620.07

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio

Por muitos anos, procurei a opção ideal entre os diversos papéis da Engenharia de Software que combinasse com minha paixão por requisitos não funcionais, onde eu tivesse exposição a todo o ciclo de vida de desenvolvimento, pudesse escrever código como se todo dia fosse uma "Hackathon" e que eu colaborasse com outras áreas da organização, exercitando conhecimento técnico, comunicação e empatia com meus colegas e clientes.

E foi assim que eu encontrei *Site Reliability Engineering* (SRE), ou Engenharia de Confiabilidade de Sites. Foi em SRE que eu encontrei o espaço para amadurecer ainda mais essas competências e implementar práticas adotadas por times de engenharia de grandes empresas que são utilizadas como referência entre os profissionais de TI.

SREs são conhecidos como "pau para toda obra", contribuindo com o desenvolvimento, testes, *releases* ou propondo mudanças de infraestrutura. SREs detêm um entendimento profundo de observabilidade (métricas, monitoramento, alertas etc.) e ajudam a organização com a melhoria de processos através do gerenciamento de incidentes, sempre envolvidos em conversas com outros times, ajudando onde for necessário.

Essa paixão por SRE é que me motivou a escrever este livro, que vai ajudar você a aprender mais sobre as práticas de SRE. Cada capítulo foi produzido com muito entusiasmo, compartilhando a experiência adquirida nos campos de batalha e descrevendo conceitos com exemplos fáceis de digerir.

Prepare um café, chá ou seja lá qual for sua bebida favorita e venha devorar estas páginas!

Ah, claro, não precisa ler tudo, apenas 99% ou 99.9% (você vai aprender mais sobre "os noves" neste livro).

Sobre o livro

Os conceitos de SRE ajudam times a obter uma disciplina para lidar com eventos inesperados, identificando as entidades envolvidas no evento e quantificando impacto através de métricas para determinar prioridades. Isso ajuda toda a organização, melhorando a percepção dos clientes e a qualidade de vida dos próprios funcionários da empresa, ao evitar o "caos" da ausência de processos, a imprevisibilidade e falta de dados sobre o estado atual do sistema e de seus usuários.

Se você está interessado(a) em se tornar SRE, ou já atua como tal mas procura saber mais sobre as implementações de tais práticas em outras organizações, vai encontrar muitas informações valiosas neste livro. Se você trabalha com Desenvolvimento, Suporte, Sistemas, QA, Análise de Dados, Operações ou qualquer outra área da Engenharia de Software, já que as práticas de SRE englobam toda a organização, você eventualmente terá que interagir com o time de SRE. Se a organização em que você trabalha não possui um time de SRE, talvez outro time ou um grupo de engenheiros(as) já esteja executando práticas similares. Então você pode obter um melhor entendimento sobre tais práticas para enriquecer essa interação e talvez você deva até recomendar este livro para eles.

Os capítulos mais conceituais deste livro são úteis para qualquer membro de uma organização que pode estar envolvido(a) em um evento onde o sistema não funciona como esperado (tal evento é conhecido como "incidente", que será explicado melhor no capítulo 2). O envolvimento não está relacionado com *culpa*, mas sim com

colaboração entre diferentes departamentos como Gerenciamento de produto ou Marketing. Por exemplo, se alguma funcionalidade do website ou aplicativo da empresa ficar indisponível por mais de uma hora, talvez seja a hora de emitir uma mensagem notificando clientes de que "Estamos enfrentando dificuldades e vamos fornecer mais informações em breve". Depois que o sistema for restaurado, praticamente todos os departamentos são mais do que bem-vindos a participar de um *Post Mortem* (a autópsia do incidente, discutida em detalhe no capítulo 12), já que nessa reunião podem ser discutidos tópicos como "impacto financeiro" ou até mesmo estratégias para acomodar as próximas campanhas de marketing.

Com a disciplina da definição de prioridades (Objetivos de Nível de Serviço são explicados no capítulo 9), este livro se torna útil também para gerentes e executivos interessados em entender os benefícios de SRE, caso estejam explorando a ideia de criar um time para melhorar a reputação do negócio da empresa através de sistemas com alta disponibilidade e amadurecer processos para, em caso de incidentes, restaurar a experiência de seus clientes de forma mais eficiente.

Basicamente, o livro é recomendado principalmente para engenheiros de software, mas os conceitos encontrados aqui devem interessar também a líderes que desejam promover um melhor equilíbrio entre a performance de um sistema e a carga de trabalho de seus engenheiros, e até mesmo colaboradores de qualquer departamento que desejam ajudar com a melhoria de processos e aprender mais sobre tecnologias relacionadas a confiabilidade.

Afinal, estamos todos juntos nisso, não é mesmo?

Como o livro está organizado

Temos um material preparado com muita dedicação para você apreciar. Vamos mergulhar no mundo da Engenharia de Confiabilidade de Sites: aprender os conceitos, falar sobre a infraestrutura que abriga os sistemas, como são observados

(Observabilidade), SLOs (Objetivos de Nível de Serviço), gerenciamento de incidentes, automação de tarefas repetitivas, Engenharia do Caos e outras práticas comuns aplicadas na indústria.

A primeira parte foca nos fundamentos. Vamos definir o que é SRE e entender como se relaciona com outros papeis na Engenharia de Software, descrever brevemente as suas competências (que têm seus respectivos capítulos onde podemos nos aprofundar depois) e vamos explicar o ciclo de vida do desenvolvimento de software.

Na parte 2 (Infraestrutura), os capítulos vão explorar arquiteturas de software confiáveis, incluindo alguns exemplos de arquiteturas utilizadas com frequência na indústria, e plataformas de runtime (execução). Vamos discutir os fundamentos do ciclo de desenvolvimento, seguido do entendimento da infraestrutura e suas tecnologias com foco em Escalabilidade, Disponibilidade e Resiliência.

A parte 3 vai mergulhar em conceitos de observabilidade, tecnologias de Métricas, Logs, Traces, ferramentas de monitoramento, e vamos desenvolver uma aplicação de exemplo utilizando a linguagem de programação Golang.

Objetivos de Nível de Serviço (*Service Level Objectives*, ou "SLOs") são apresentados na parte 4, este é o conceito mais importante para SREs já que guia todas as outras práticas. SLOs ajudam toda a organização a identificar como um determinado serviço se comporta, a severidade de sua interrupção e como observar seus principais indicadores. Se você tiver que priorizar alguma parte deste livro para a sua leitura, estes são os capítulos que você está procurando.

As competências de SRE serão explicadas com mais detalhes na parte 5 (Gerenciamento de incidentes). Munidos do conhecimento adquirido nas partes anteriores, podemos imergir no ciclo de vida de um incidente e *Post Mortems* e, quando analisarmos um exemplo de um incidente, poderemos reconhecer os conceitos e tecnologias que serão mencionados. Vamos explorar outros tópicos da Engenharia de

Confiabilidade, discutindo oportunidades para automatizar atividades repetitivas (*eliminate toil*) e desenvolver ferramentas para ajudar a resolver tarefas rotineiras.

Vamos conhecer a Engenharia do Caos e aprender como SRE pode colaborar com outros engenheiros para realizar "injeções de falhas" em serviços, assim como seus respectivos componentes de infraestrutura ou dependências, para, propositalmente, criar eventos que testam a resiliência dos serviços e aprender como ele se comporta durante um desastre ou uma degradação parcial. Vamos falar também sobre outras práticas comuns da indústria como monitoramento sintético, interruptores de funcionalidades (*feature toggles*), redirecionamento de tráfego para resolução de incidentes e algumas dicas sobre como gerenciar a confiabilidade de aplicativos móveis.

Espero que o aprendizado deste livro sirva como combustível para discussões voltadas para a adoção de novos processos e que ajude leitores a navegar o planejamento de novas propostas para Observabilidade e Gerenciamento de Incidentes.

Sobre o autor

Natural de Fortaleza, Ceará, Marcelo Costa atua como Site Reliability Engineer em Chicago, Illinois. Começou sua carreira como Desenvolvedor Web enquanto estudava para obter o diploma de Bacharel em Administração de empresas. Logo depois aceitou uma proposta para se tornar Engenheiro de Suporte, trabalhando para uma multinacional responsável por soluções de gerenciamento de conteúdo com uma suite de produtos compatível com várias tecnologias de gigantes da indústria como Microsoft e Oracle. Teve exposição a diferentes arquiteturas, problemas de performance e investigações de *bugs*. Obteve várias certificações, entre elas: SCJP (*Sun Certified Java Programmer*), LPIC (*Linux Professional Institute Certified*), ITIL (*Information Technology Infrastructure Library*) e FCE (*First Certificate in English*) emitida pela *University of Cambridge ESOL* (*English for Speakers of Other Languages*).

Em 2012, se mudou para a Irlanda para trabalhar como *Senior Sytems Engineer* e obteve um mestrado em *Enterprise Application Development* na *Atlantic Technological University* em *Letterkenny, Co. Donegal, Ireland*. Trabalhou em empresas famosas como IBM e Twitter. Nesse período, como *Cloud Integration Engineer*, esteve envolvido em projetos de pipelines de Integração Contínua e Entrega Contínua de serviços conteinerizados com a tecnologia *Docker* para *deployments* com *Kubernetes* em plataformas na Nuvem. Depois, como *Reliability Engineer*, utilizou ferramentas de monitoramento e gerenciou incidentes no TCC (*Twitter Command Center*) como IMOC (*Incident Manager On-Call*).

Em 2019, se mudou novamente, desta vez para os Estados Unidos para trabalhar na Universidade de Chicago e atualmente trabalha para o setor privado como *Site Reliability Engineer*, envolvido em projetos de observabilidade e automação para gerenciamento de

incidentes para melhorar a confiabilidade de serviços operando na plataforma *Amazon Web Services* (AWS).

O autor teve experiência com ambientes onde as práticas de SRE já eram bastante amadurecidas e atualmente trabalha em projetos para a implementação das mesmas práticas em seus estágios mais iniciais.

Agradecimentos

Eu agradeço aos meus pais: Paulo e Ozélia, pelo amor, inspiração e por serem meus melhores amigos. Karol, minha parceira nessa louca vida de um quase-nômade-digital. Aos meus filhos Heitor, Bela (Humanos), William, Feijoada e Bruno (Furões). Nada disso teria sentido sem eles.

Ao meu irmão Bruno, minha bússola para a vida, com sua inteligência, eloquência e humor. Muito grato também a Luíza, Cecília, Maurício, Naiana e todos meus parentes por todo o suporte familiar.

Sou eternamente agradecido a Vivian Matsui por ter acreditado na minha proposta do livro, pela revisão tão profissional e pelo "coaching" no decorrer desta jornada como autor. Agradecimentos especiais para Wilson Marçal, Ana Cavalcante e Ana Aguiar por ajudarem com a melhoria do conteúdo.

I would like to leave special mentions to: Kyler Bush, an awesome person that assembled a kick-ass SRE team, Andrew Prokhorenkov for being such a kind empathic genius and Andrejus Chaliapinas, a great friend with incredible tech leadership skills.

Queria agradecer também a todos os amigos do Brasil, da Irlanda e os de Chicago.

Parte 1: Fundamentos

Capítulo 1
O que é SRE?

1.1 Conhecendo SRE

Site Reliability Engineering, ou Engenharia de Confiabilidade de Sites, é a disciplina que utiliza conhecimento de Engenharia de Software para implementar melhorias na infraestrutura e nas operações de uma organização, buscando o melhor uso de tecnologia, automação e aperfeiçoamento de processos para lidar com incidentes e evitar que os clientes de um determinado sistema sofram um impacto substancialmente negativo.

SRE é geralmente explicado através de um exercício de pensamento: "e se um time de desenvolvimento de software ficasse responsável por automatizar um time de operações?". Para refletir melhor sobre essa indagação, temos que entender o que são times de desenvolvimento e de operações.

O time de desenvolvimento de uma organização que trabalha com Engenharia de Software é responsável por identificar especificações de clientes, elaborar uma solução para satisfazê-las (*design*) e implementar tal solução, produzindo código e colaborando com outros times para entregá-la ao cliente.

O time de operações, entretanto, atua atrás das cortinas, mas possui muitas atividades importantes em seu escopo:

- Manter inventário de recursos de infraestrutura e um catálogo de aplicações (também conhecidos como serviços).
- Gerenciar configuração e a instalação do software em uma plataforma de execução (*deploy*) e lançar novas versões de serviços para o usuário final (*release*).

- Monitorar a infraestrutura e observar o sucesso dos serviços que compreendem o sistema enquanto ele é utilizado por clientes.
- Participar de um revezamento no qual um membro do time fica responsável por receber notificações de alertas, que são iniciados quando problemas no sistema são detectados, e reagir a tais chamados para resolver incidentes (*on-call*).
- Administrar documentação para o suporte de serviços (*runbooks* ou *playbooks*), incluindo ações que devem ser tomadas para os respectivos alarmes e eventos associados com a degradação de serviços.
- Fornecer manutenção para a infraestrutura gerenciando backups e aplicando atualizações quando necessário.
- Colaborar com pessoas desenvolvedoras, engenheiros de testes (*Quality Assurance* ou apenas QA), profissionais de Segurança da Informação (*Cybersecurity* ou *InfoSec*), gerentes etc., para buscar melhoria contínua para processos do ciclo de vida do desenvolvimento de software.
- Outras atividades como: gerenciar contratos com terceiros, outras organizações que fornecem software ou infraestrutura (por exemplo: software de monitoramento ou soluções na nuvem).

Portanto, SRE surgiu através da ideia de alocar um time de desenvolvimento, colaborando com engenheiros de infraestrutura com experiência em operações, para mapear as tarefas operacionais do ciclo de vida do desenvolvimento de software (catálogo de serviços, gerenciamento de ambientes, deploy, release, on-call, runbooks etc.), implementar soluções e instaurar novos processos para colaborar com outros times, melhorando a confiabilidade de seus serviços.

1.2 Um pouco de história

O primeiro time de SRE surgiu em 2003 no Google, a empresa que criou o conceito, devido a suas necessidades de melhorar a confiabilidade de seus serviços já que, na época, a experiência de usuários na internet e a expectativa do mercado se transformava, exigindo que sistemas não ficassem indisponíveis por muito tempo. A indisponibilidade (*downtime*) de um sistema reflete em uma má reputação, afeta a percepção da qualidade da organização, o que afasta clientes, investidores e parceiros.

O Google teve tanto sucesso com a elaboração desta prática que lançou livros compartilhando os conceitos de SRE: *Site Reliability Engineering: Como o Google Administra Seus Sistemas de Produção* e *The Site Reliability Workbook: Maneiras Práticas de Implementar SRE*, que podem ser encontrados neste link: https://sre.google/books/. Existem outros livros interessantes como o *Implementing Service Level Objectives* (de Alex Hidalgo) e *Observability Engineering* (de Charity Majors, Liz Fong-Jones e George Miranda), que são obras que mergulham mais em conceitos que auxiliam na implementação de SRE.

Então você deve estar se perguntando: "Qual é a proposta deste livro?". Neste livro nós vamos explicar conceitos de SRE do nosso jeito. Vamos sintetizar a literatura existente para ajudar você a digerir esses conceitos de uma forma amigável, compartilhando experiências reais de gigantes da indústria, onde as práticas de SRE já são bastante amadurecidas, e até discutindo como começar a implementar SRE em uma empresa do zero, ajudando você a produzir propostas para a liderança da organização, evangelizando as práticas de SRE e educando colegas sobre seus benefícios.

1.3 Outros times de engenharia de software e SRE

Para elaborar melhor como o SRE se relaciona com outros times de engenharia de software, vamos definir brevemente como as responsabilidades de cada time se encontram.

O/A desenvolvedor(a) que escreve o código das aplicações (serviços) geralmente foca nas funcionalidades que atendem a necessidade do cliente (*functional features*), enquanto Engenheiros(as) de Operações focam na parte não funcional (*non-functional features*) que compreendem requisitos operacionais: escalabilidade, disponibilidade, extensibilidade, consistência, resiliência, usabilidade, observabilidade, segurança, durabilidade e agilidade nos processos para uma melhor manutenção. Historicamente, essa diferença fundamental entre os focos de cada time, Desenvolvimento (*Dev*) e Operações (*Ops*), tem criado um atrito no processo de entrega de mudanças, isto é, de novas versões, de software, o que levou ao advento da prática de DevOps.

DevOps almeja melhorar a colaboração entre esses times, utilizando automação para melhorar a eficiência de cada passo do ciclo de vida do desenvolvimento de software. Livros como *The Phoenix Project* e *The Unicorn Project* são "romances DevOps" bem conhecidos que exemplificam como esses atritos afetam o negócio da empresa e como superá-los.

Isso traz uma pergunta comum: "Então qual é a diferença entre DevOps e SRE?".

Em seu conceito original, DevOps é um paradigma, não deve ser um "time". Porém, na prática, muitas empresas decidem criar tal time para alocar profissionais que vão focar mais incisivamente na melhoria da experiência de desenvolvimento e *releases* de software.

O SRE, entretanto, é mais abrangente em termos de impacto organizacional e tem um processo de implementação bem mais específico. SRE vai além da ideia de conectar dois times existentes (Dev e Ops), o escopo de SRE inclui a introdução de conceitos como

SLA (Acordo de Nível de Serviço), SLO (Objetivos de Nível de Serviço) e Orçamento de erros (Error Budget), o Gerenciamento de Incidentes, revezamentos On-Call da empresa, reuniões de *Post Mortems*, simulações de incêndio (*Fire Drills*), colaboração com times de desenvolvimento para melhorar a resiliência de seus serviços e estratégias de releases, entre outras atividades. (Vamos nos aprofundar mais em todas essas competências de SRE no próximo capítulo).

Profissionais de SRE têm um currículo bem diversificado, com experiência em Suporte, Desenvolvimento, Operações, Redes, Testes (QA), Administração de Sistemas e Infraestrutura.

E aí, despertei seu interesse? Vamos aprender mais sobre SRE nos próximos capítulos!

Capítulo 2
Competências de SRE

2.1 SLOs, SLIs e Orçamento de Erro

Você já visitou um website que parecia meio lento mas, logo depois de atualizar a página, começou a responder normalmente? Ou algum aplicativo móvel que apresentou uma mensagem de erro e ela desapareceu quando você tocou novamente no mesmo botão? Esses são exemplos de situações em que algum sistema ou equipamento não funciona como esperado, mas quando o usuário, naturalmente, tenta a mesma funcionalidade novamente, obtém o resultado esperado.

Isso faz parte do etos de SRE: não se deve esperar que algo desempenhe suas funções perfeitamente o tempo todo. Em outras palavras: um sistema não deve operar com 100% de confiabilidade. Não é realístico ou mesmo estratégico que uma organização não permita eventuais interrupções em seus serviços, portanto é comum definir um Objetivo de Nível de Serviço, ou *Service LeveL Objective* (SLO), alinhado com as circunstâncias do sistema, a infraestrutura e pessoas que trabalham com sua manutenção. Tais objetivos são expressados através de múltiplos "noves", por exemplo, 99.9% ou 99.999%, que definem o tempo em que se espera que um sistema realize suas funções corretamente.

Cada serviço deve possuir um SLO, que é definido e ajustado por SREs de forma colaborativa com os(as) engenheiros responsáveis. O SLO é violado quando o serviço não opera de acordo com as condições de seus "Indicadores de Nível de Serviço", ou *Service Level Indicators* (SLIs). Em outros termos, os SLIs são condições e o SLO é um "contrato" que expressa o tempo em que todas as condições devem ser cumpridas. Quando o serviço falha em

satisfazer as condições de um ou mais SLIs além do tempo permitido, o SLO é violado.

O Orçamento de Erro é o tempo concedido para um serviço operar sem satisfazer suas condições de confiabilidade (SLIs). É basicamente o número de violações de SLO que podem ser toleradas antes que o problema torne-se crítico o suficiente para afetar o negócio de maneira substancial.

Para ilustrar, vamos utilizar como exemplo um website X que possui os seguintes SLIs:

- Os clientes devem visitar o website sem encontrar erros.
- A página inicial deve ser apresentada para os usuários em até 400 milissegundos (ms).

O primeiro SLI é baseado na condição de que um usuário deve acessar o website e suas páginas devem ser carregadas no navegador com sucesso; o segundo SLI é baseado no tempo de resposta, ou latência, do website. Vamos assumir que, por exemplo, o SLO para este serviço (website X) é 99.9%. Isso quer dizer que podemos expressar essas condições da seguinte maneira:

- 99.9% das requisições direcionadas ao website não devem retornar erros (Código de resposta HTTP != 5xx).
- 99.9% das requisições direcionadas à página inicial do website devem ser processadas em até 400ms (tempo de resposta < 400ms).

Por fim, se inserir o Orçamento de Erro, considerando o SLO 99.9%, temos uma maneira alternativa de expressar as mesmas condições:

- Permitir que apenas 0.1% das requisições direcionadas ao website retornem erros (Código de resposta HTTP == 5xx).
- Permitir que apenas 0.1% das requisições direcionadas à página inicial do website sejam processadas em mais de 400ms (tempo de resposta > 400ms).

Supondo que as métricas desse website sejam emitidas, agregadas em um banco de dados de série temporal (vamos discutir isso melhor no capítulo 6), e que os dados dessas métricas sejam disponibilizados a cada minuto, então as condições podem ser ainda mais específicas. Um SLO de 99.9% define um Orçamento de Erro de 0.1% que, em termos de "minutos bons" e "minutos ruins", corresponde a 10 minutos ruins por semana ou 43 minutos ruins por mês que estão à sua disposição no orçamento. Então podemos reescrever as condições novamente:

- Permitir apenas 10 minutos ruins em que a taxa de sucesso das requisições ao website fique abaixo de 99.9% OU que o tempo de reposta das requisições à página inicial do website seja maior do que 400ms no decorrer de uma semana.

Esse tipo de matemática é comum no dia a dia de SREs e existem muitas ferramentas online para calcular tempos de disponibilidade (*uptime*), por exemplo, a https://uptime.is.

Isso é apenas uma introdução, nós vamos estudar os conceitos de SLO, SLIs e Orçamento de Erro com mais detalhes e exemplos práticos no capítulo 9.

2.2 Gerenciamento de incidentes

O gerenciamento de incidentes é uma das competências mais importantes para SRE. Para entender melhor vamos, primeiro, nos familiarizar com alguns conceitos básicos:

- **Incidente:** um incidente é um evento não planejado que afeta a funcionalidade de um serviço. Tal evento faz com que o sistema não se comporte da maneira esperada impactando um número significante de usuários. Um incidente é algo que requer atenção *imediata* e precisa de uma resposta *coordenada*.

- **Serviço:** serviços são os componentes de sistemas maiores que formam o "produto" ou, em outras palavras, as funcionalidades oferecidas para os usuários. Um serviço que produz erros, ou que manifesta problemas de performance, impacta um "Cliente". O Cliente não é, necessariamente, o usuário final de um sistema; um aplicativo que recebe uma resposta ruim de outro aplicativo também é considerado um cliente que é impactado negativamente. Porém, esse mesmo evento pode não ser imediatamente propagado até os usuários de um website ou um aplicativo móvel, por exemplo.

Normalmente, quando algum serviço não opera da maneira esperada, SREs são as primeiras pessoas na "cena do crime". Engenheiros(as) de SRE participam de um revezamento onde recebem notificações quando algum incidente em potencial ameaça a estabilidade do sistema. Tais notificações são nostalgicamente conhecidas como "pages", em referência ao "Pager", que é um dispositivo antigo utilizado para entrar em contato com profissionais em situações de urgência.

Porém, nem todo evento que afeta um serviço necessariamente gera um novo incidente, isso fica ao critério de Engenheiros(as) SRE, que devem tomar decisões baseadas na triagem do evento ou consultando especialistas e outras pessoas engenheiras que têm conhecimento mais minucioso sobre o sistema e o impacto ao negócio.

Quando incidentes são iniciados, SREs são nomeados "Comandantes de Incidente" (*Incident Commanders*) e ficam responsáveis, entre outras tarefas, por:

- Avaliar impacto (triagem) uma vez que o problema é detectado.
- Organizar um canal de comunicação onde o problema deve ser examinado.

- Entrar em contato com engenheiros de escalação e definir os investigadores principais para auxiliar no diagnóstico.
- Coordenar ações corretivas (colaborando com o resto do time de engenharia para buscar uma mitigação).
- Agendar e conduzir reuniões de *Post Mortem*.

O ciclo de vida de um incidente é explicado com mais detalhes no capítulo 12.

2.3 Observabilidade

SREs são os guardiões do negócio, constantemente observando todos os sistemas, seus serviços, infraestrutura e eventos de uma organização, mas essa observação só é possível com telemetria. O que é telemetria? É a combinação das palavras gregas "tele", que significa "remoto", e "métron" que quer dizer "algo usado para medir". A telemetria representa qualquer dado que é emitido por um serviço para ser acessado ou visualizado remotamente para fins de monitoramento.

E o que é observabilidade? É quando o estado ou a saúde de um sistema complexo pode ser entendido baseado apenas na telemetria emitido por ele.

Os três pilares de observabilidade são: métricas, logs e rastros (*traces*). Métricas são dados que, em sua forma mais simples, compreendem nomes e valores numéricos que podem ser utilizados para verificações quantitativas (exemplo: `percentual_de_uso_de_cpu` ou `numero_de_processos`). Logs são mensagens emitidas por uma aplicação para auxiliar em seu diagnóstico (exemplo: `2023-09-23T22:47:39.863Z info service/service.go:148 Everything is ready. Begin running and processing data`). Rastros (*traces*) representam a jornada de uma transação através de diferentes pontos (*spans*) de um sistema (exemplo: um rastro pode informar que uma requisição `HTTP`

`GET /banco/conta/saldo` levou `400ms` para ser processada e ele inclui cada passo dessa operação, suas respectivas funções e tempo de execução).

Ao navegarmos pelos capítulos 6, 7 e 8, vamos conhecer melhor os três pilares de observabilidade e vamos desenvolver uma aplicação de exemplo para brincar com cada uma dessas funcionalidades.

2.4 Desenvolvimento de ferramentas e automação de atividades repetitivas

Pessoas que atuam como SRE estão sempre buscando "automatizarem elas mesmas" para que operações de manutenção de software não consumam seu tempo e que tarefas repetitivas não sejam suscetíveis a erros. O *Livro de SRE* (Murphy, 2016), em seu capítulo 5: "Eliminando o trabalho pesado" (em inglês: *Eliminating toil*), possui uma descrição do que é considerado ***toil***:

> *Toil* é o tipo de trabalho associado à execução de um serviço que é manual, repetitivo, automatizável, tático, que não possui valor duradouro e que escala linearmente à medida que o serviço cresce.

Portanto se um(a) SRE se encontra realizando uma operação como, por exemplo, executar um script para restaurar a condição de um serviço sem alterar sua performance (sem trazer valor duradouro), e tal tarefa é realizada continuamente de uma forma que poderia ser orquestrada por outro evento – em outras palavras, onde existe uma possibilidade para introduzir automação –, diz-se que essa operação é considerada *toil*. O *toil* consome tempo que poderia ser aplicado em atividades de engenharia, e lembre-se que SREs são engenheiros(as) que automatizam operações.

A ideia é manter a Engenharia de Software de uma organização mais escalável já que o número de tais atividades, se precisarem ser aplicadas para cada serviço, vai aumentar proporcionalmente à medida que novos serviços são criados. Este mesmo raciocínio aplica-se a atividades operacionais necessárias para novos times, novos componentes de infraestrutura, novos *data centers* etc.

SREs são responsáveis por reagir a um evento inesperado e conduzir uma triagem: identificando o(s) serviço(s) impactado(s), analisando dados, verificando documentação, *deployments* realizados recentemente, praticamente todas as informações disponíveis sobre os componentes do sistema que apresentam problemas. Essa tarefa, como você pode ver, assume que tais informações existam e estejam acessíveis.

Imagine um cenário em que um(a) engenheiro(a) de SRE recebe um "page" as 3 da manhã: "Serviço X está reportando alta latência". Ele(a) procura uma documentação sobre o serviço entre os Wikis disponibilizados na intranet da empresa mas não consegue encontrar nenhuma informação, nada de métricas ou logs, nenhum time de escalação atribuído a este serviço, não encontra nenhuma notificação sobre algum *deployment* recente no chat da empresa e nem mesmo o repositório de código que pertence ao serviço X.

Nesse cenário, como podemos ver, é extremamente desafiador para a pessoa que está de plantão (do inglês *on call*) fornecer suporte para este serviço X. Será necessário um esforço substancial para procurar algum documento técnico que descreva a funcionalidade do serviço assim como a leitura de seu código, configuração, e a busca por qualquer telemetria que está sendo emitida por este serviço. A ausência dessas informações torna o trabalho de SRE muito complicado já que o *tempo* é um recurso muito importante quando precisamos recuperar algum sistema que não está funcionando como esperado, pois quando o sistema fica instável por muito tempo, aumenta-se o número de usuários frustrados que abandonam um website ou aplicativo móvel e procuram a concorrência.

Tal cenário ainda inclui o risco de o serviço ser associado com uma funcionalidade de baixa prioridade, onde transações lentas ou até mesmo a interrupção completa de seu trabalho computacional seriam aceitáveis por várias horas. Então quem está *on call* pode, potencialmente, investir muito tempo durante a madrugada investigando um problema que, no final das contas, não era tão crítico assim e poderia ser investigado na manhã seguinte com o time responsável pelo serviço X. Isso prova que as informações relacionadas ao serviço são essenciais para permitir que alguém reaja a eventuais alarmes de forma eficiente, não apenas mitigando um incidente para restaurar a experiencia dos usuários rapidamente, munidos da documentação e telemetria necessária, mas também melhorando a qualidade de vida de engenheiros(as) que estão *on call*.

Portanto, em vez de apenas reagir a alarmes e potenciais incidentes, SREs também devem verificar se serviços estão prontos para operar em um ambiente de *produção* antes mesmo de qualquer *release*. Algumas organizações até criam regras internas em que um determinado time não pode lançar uma nova versão de um serviço até que ele obtenha uma pontuação adequada baseada em requisitos operacionais. SREs colaboram com o resto da engenharia para determinar essa lista de requisitos e qual deve ser o *score* mínimo para que um serviço seja "certificado" como *production-ready*.

Por consequência, SREs buscam melhorar a "encontrabilidade" de serviços e seus artefatos (em inglês, o termo utilizado é: *discoverability*), pois essas informações são essenciais para a triagem de alarmes e um gerenciamento de incidentes eficaz.

Vamos discutir o desenvolvimento de ferramentas no capítulo 11 e discutir cenários com oportunidades para automação. Este mesmo capítulo também fornece um exemplo prático sobre requisitos operacionais que devem ser verificados para atestar que um serviço está *production-ready*, apresentando sugestões de design e descrevendo como tal automação pode ser implementada.

2.5 Engenharia do Caos

Na Engenharia do Caos, SREs planejam *simulações de incêndio* e, além de explorar maneiras de verificar a resiliência dos serviços, tais simulações também ajudam a testar os *processos* envolvidos no fluxo esperado de um incidente: detecção de um problema, triagem, exame, diagnóstico, mitigação e resolução. É uma atividade importante para desenvolver uma "memória muscular" em que os membros de um time de escalação possam responder a eventuais desastres sem entrar em pânico ou ficarem perdidos sem saber qual é o próximo passo.

A atividade de planejar *injeções de falhas* pode ser bastante divertida e envolve um aprendizado significativo sobre o sistema e sua arquitetura, mas até uma simples notificação anunciando um falso incidente, mesmo sem causar uma degradação na performance de um serviço, pode revelar atritos no processo ou até mesmo a ausência de artefatos como documentação, telemetria etc.

As simulações de incêndio são discutidas no capítulo 13.

2.6 O que mais SREs fazem?

Descrevemos neste capítulo algumas das competências de SRE, mas a Engenharia de Confiabilidade de Sites pode estender o seu escopo de acordo com as necessidades da organização, navegando por diferentes pontos do ciclo de desenvolvimento de software. SREs podem colaborar com:

- QA e SDETs para trabalhar com pipelines, desenvolver testes automatizados e gerenciar _*releases*.
- DevOps para ajudar no desenvolvimento de ferramentas de *deployment* e "plugar" mais mecanismos de observabilidade.

- Engenheiros(as) de Cibersegurança (DevSecOps), pois possuem pontos em comum com o processo de gerenciamento de incidentes, incluindo práticas de prevenção utilizando automações.
- Times de desenvolvimento para melhorar o amálgama de *features* funcionais e não funcionais, aperfeiçoando a confiabilidade dos serviços.
- Gerentes, Diretorias etc. SREs podem utilizar sua expertise com observabilidade para auxiliar na obtenção e visualização de dados que auxiliam no monitoramento de indicadores importantes para o negócio da empresa.

Isso implica que SREs têm entre suas competências a aptidão para explorar novos processos e encontrar oportunidades para implementar as práticas de confiabilidade (Acordos de Níveis de Serviços, Observabilidade, Gerenciamento de Incidentes etc.) e uma excelente habilidade de comunicação, já que a jornada de SRE envolve colaboração desde o nível mais técnico envolvendo mudanças de código e arquitetura de software, até a discussão com a liderança de times para evangelização de novos processos ou relatórios de *Post Mortems*, incluindo detalhes sobre o impacto de negócio, que têm como audiência a camada executiva da empresa.

O propósito de SRE é "manter as luzes acesas" e buscar melhoria contínua, portanto os engenheiros(as) de confiabilidade de sites devem estar preparados para aprender novos processos e tecnologias para colaborar com quaisquer agentes ou departamentos da organização.

Capítulo 3
O ciclo de vida do desenvolvimento de software

Conhecer o ciclo de vida do desenvolvimento de software de uma organização é essencial para mapear seus padrões predominantes de engenharia: as ferramentas, o comportamento das pipelines de testes, a cadência de entrega de novas versões, a infraestrutura, os ambientes e, além do contexto mais técnico, entender as dinânicas "sociotecnológicas" da colaboração entre times.

Como visto no capítulo anterior, SREs observam a organização de uma maneira holística e interagem com outros times para discutir melhorias, não apenas de software (funcionais e não funcionais), mas melhorias de processos. Portanto, SREs devem se familiarizar com boas práticas de gerenciamento de ambientes, infraestrutura, *pipelines*, *releases* e *deployment*, estendendo seu escopo além da área de operações e manutenção de serviços em produção. Esse conhecimento ajuda a SRE a realizar triagens de maneira mais intuitiva e enriquece a discussão durante *Post Mortems*.

3.1 Ambientes

Para explorarmos este assunto, vamos primeiro falar sobre "Ambientes". Quando desenvolvedores(as) começam a trabalhar em algum código novo, alguma mudança de código para ajustar uma funcionalidade existente ou mesmo para consertar alguma regressão (um *bug* na aplicação), o código é geralmente testado localmente em suas máquinas ou em um ambiente de desenvolvimento, também conhecido apenas como *Dev*. Quando confirmam que a mudança de código faz com que a aplicação execute o novo comportamento esperado, a mudança é "promovida" para o

ambiente de *Stage* (ou *Staging*) para que as funcionalidades de múltiplos serviços possam ser testadas em conjunto com outras mudanças criadas por outros desenvolvedores(as) em um ambiente mais estável. Uma vez que os testes são executados com sucesso, a mudança é promovida mais uma vez para um ambiente de *Pré-produção* (*PreProd*), que deve replicar a configuração e a infraestrutura do ambiente de *Produção* da forma mais aproximada possível. Essa é uma prática comum para evitar surpresas devido à ausência de paridade de configuração entre ambientes, portanto essa nova versão da aplicação é verificada mais uma vez antes que a mudança de código seja finalmente promovida para *Produção* (ou apenas *Prod*), que é o objetivo final: entregar uma nova funcionalidade para o ambiente que é exposto aos usuários.

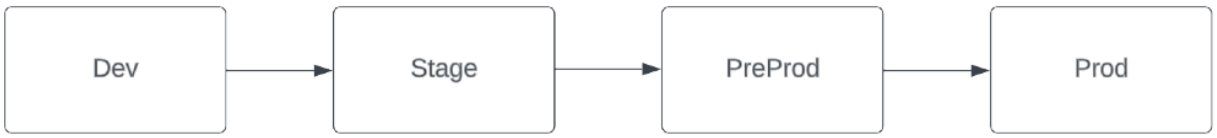

Figura 3.1: Ambientes e o fluxo da promoção de mudanças de código

Note que essa é apenas a descrição de um ciclo comum. Algumas organizações definem apenas 3 ambientes (*Dev*, *Stage* e *Prod*), ou utilizam nomes diferentes para o ambiente de *Stage* como QA (*Quality Assurance*) ou UAT (*User Acceptance Testing*), e até mesmo provisionam um ambiente adicional de *Pós-produção* para testar códigos para correção de bugs críticos conhecidos como *"hot fixes"*, que devem ser testados em um ambiente semelhante a *Produção*, mas sem impactar o ciclo de desenvolvimento que está em andamento nos outros ambientes.

Todos esses ambientes abrigam os serviços que compõem o produto e precisam ser gerenciados para facilitar o ciclo de desenvolvimento. Ambientes de desenvolvimento (*Dev*) são vitais para que as mudanças de código sejam implementadas; ambientes de testes (*Stage*) devem estar disponíveis para evitar atrasos no processo de

QA; ambientes de *Pré-produção* também precisam de manutenção para manter configurações e base de dados semelhantes às que se encontram em ambientes que são expostos aos usuários; e, finalmente, o ambiente de *Produção* deve ser monitorado e modificado com mais cautela, tanto seus serviços quanto sua infraestrutura. O foco de SRE está, definitivamente, no ambiente de *Produção*. Algumas empresas como a Meta, por exemplo, nomeiam seu time de SRE como "Production Engineering" e SREs como "Production Engineers".

Ambientes de desenvolvimento

Embora SREs foquem seus esforços no ambiente de *Produção*, engenheiros(as) de confiabilidade devem munir-se de conhecimento relacionado a boas práticas operacionais e colaborar com DevOps para utilizar melhor os ambientes de desenvolvimento. Afinal de contas, se o ciclo de vida do desenvolvimento se inicia no ambiente de *Dev*, este ambiente é crítico para que SREs criem estratégias alinhadas com a cultura de falhar rápido (*fail fast*). Quando o(a) desenvolvedor(a) cria um serviço novo ou inicia sua jornada para implementar ou modificar uma funcionalidade de um serviço existente, SREs podem colaborar com desenvolvedores(as) para se certificarem de que alguns pré-requisitos operacionais básicos, como Monitoramento, por exemplo, estejam incluídos no berço dessa mudança.

Um time de desenvolvimento tem múltiplos engenheiros e engenheiras trabalhando em mudanças de código em paralelo, portanto a utilização de um único ambiente de desenvolvimento pode introduzir um atrito na produtividade do time. Uma solução para este problema é o uso de ambientes "caixa de areia" (*sandbox*) exclusivos para cada desenvolvedor(a), onde o código pode ser testado sem interferência de outras mudanças que ainda não estão estáveis o suficiente. Uma prática comum na indústria envolve a criação de um conjunto de microambientes de desenvolvimento (*pool of sandboxes*) que podem ser alocados temporariamente para

um membro do time de desenvolvimento e, uma vez que sua atividade de desenvolvimento for concluída, o mesmo ambiente de *sandbox* possa ser recriado, apagando todos os dados e reprovisionando sua infraestrutura para que os próximos desenvolvedores possam utilizá-lo.

Existem várias maneiras de gerenciar ambientes de desenvolvimento, mas SREs devem focar em se certificar que os mecanismos e a infraestrutura que ajudam a monitorar e proteger serviços em Produção estão disponíveis também nos ambientes de Desenvolvimento. Assim os desenvolvedores podem observar métricas e testar interruptores de funcionalidades (*feature toggles*) o mais cedo possível no ciclo de desenvolvimento, o que é bastante útil para conduzir simulações de desastres em outros ambientes de *não produção* no futuro.

Além dos desafios relacionados aos ambientes de desenvolvimento, ambientes de *Stage* e *Pré-produção* também têm suas peculiaridades. Eles precisam permanecer estáveis o suficiente para acomodar testes, do contrário a nova versão do software não poderá ser promovida para *Produção*. A falta de paridade de configuração entre ambientes de *não produção* e *Produção* é um problema comum que tem um grande potencial para causar incidentes, por isso recomenda-se que a diferença entre os ambientes deve ser minimizada. Esta recomendação é mencionada na metodologia "12 (Twelve) Factor App" (https://12factor.net/dev-prod-parity) como um dos 12 fatores que tornam uma aplicação mais moderna e confiável.

Gerenciamento de ambientes

Devido à importância do gerenciamento de ambientes, algumas organizações definem times focados em melhorar a experiência de desenvolvimento e investem na metodologia de "Infraestrutura como Código" (*Infrastructure as Code*, ou apenas IaC) para automatizar o provisionamento da infraestrutura e o *deployment* de serviços para gerar ambientes novos a qualquer momento. Para os ambientes de desenvolvimento, por exemplo, os ambientes de *sandbox* podem ser

inicializados localmente com máquinas virtuais (*Virtual Machines*, ou apenas VMs) ou contêineres utilizando ferramentas como Vagrant ou Docker. Para a gestão de ambientes, além do escopo do desenvolvimento local, existem ferramentas de Gerenciamento de Configuração (*Configuration Management*) como Chef, Puppet ou Ansible, por exemplo, para automatizar a instalação de dependências e a configuração de ambientes. Para o provisionamento de infraestrutura, o Terraform é uma solução mais moderna e alinhada com provedores de serviços na nuvem (*Cloud Services*), como Microsoft Azure ou Amazon Web Services (AWS), que serão mais explorados no capítulo 5.

Com o advento das práticas de IaC e *Configuration Management*, mais times de engenharia têm adotado o uso de ambientes efêmeros (que existem por um curto espaço de tempo), já que múltiplos ambientes podem ser provisionados e destruídos a qualquer momento, tratados como "gado" ou "cattle" (em grande volume, sem identidade duradoura ou tratamento especial), ao invés de ambientes tradicionais construídos manualmente e que possuem nomes (que seriam equivalentes a "animais de estimação" ou "pets").

Esta metáfora "cattle vs. pets" é bastante referenciada no meio DevOps para orientar times de engenharia a evitar o provisionamento manual de ambientes e infraestrutura, pois eles requerem manutenção cuidadosa já que sua configuração não pode ser facilmente replicada através de automação. Se algum recurso de infraestrutura crítico como um volume de armazenamento ou uma fila de mensageria for deletado acidentalmente em produção, é muito importante possuir a habilidade de reprovisionar a infraestrutura automaticamente através de IaC e restaurar o funcionamento do sistema o mais rápido possível para a confiabilidade de sites.

Outra vantagem, em termos de escalabilidade, é a utilização de IaC e *Configuration Management* para provisionar uma réplica de um ambiente (incluindo toda sua infraestrutura) em um novo Data

Center ou região para atender melhor usuários geograficamente distribuídos. Vamos falar mais sobre infraestrutura e ferramentas de provisionamento no capítulo 5.

3.2 Uma breve revisão sobre Integração Contínua e Entrega Contínua (CI/CD)

Além de conhecer o ciclo de vida de desenvolvimento do software e como as mudanças de código viajam entre ambientes, SREs devem dominar os conceitos de Integração Contínua (*Continuous Integration*) e Entrega Contínua (*Continuous Delivery*). Pipelines de CI/CD são implementadas para otimizar e acelerar o processo de testes e entrega de novas versões de software. SREs devem conhecer esses conceitos e se familiarizar com as pipelines implementadas em sua organização para entender como as mudanças de código chegam até o ambiente de produção e discutir oportunidades para aplicar mais escrutínio se necessário.

Integração Contínua (Continuous Integration)

A pipeline de Integração Contínua (CI) tem como objetivo evitar o cenário em que algum membro do time de desenvolvimento finalize a implementação de uma funcionalidade e declare que o código pode ser promovido pois ele *funciona em sua máquina*, presumindo erroneamente que, ao executar testes localmente de maneira isolada, o aplicativo ou sistema funcionará da forma esperada em outros ambientes que contêm todas as outras mudanças e serviços modificados por outros desenvolvedores.

A Integração Contínua compreende um ou mais ambientes onde mudanças de múltiplos autores são testadas em conjunto de forma centralizada automaticamente. Abordagens comuns envolvem o *deploy* de múltiplas mudanças para um ambiente de testes que é alvo da execução de suítes de testes automatizados iniciada através

de tarefas agendadas, para certificar as últimas integrações válidas no decorrer do dia, ou a execução do mesmo procedimento envolvendo deploy e testes automatizados para cada uma das mudanças. Em outras palavras, testando cada *commit* de uma *feature branch* e iniciando a automação através de um *Pull Request*.

Um estudo do Instituto Gartner, organização dedicada a pesquisas e consultorias de tecnologia da informação, cita que 80% dos incidentes que impactam serviços críticos são causados por problemas relacionados a pessoas e processos, e que mais de 50% destes incidentes são causados por mudanças (*"Configuration Management for Virtual and Cloud Infrastructures"*, 2010).

SREs precisam estar cientes dos vários tipos de mudanças que são aplicadas no ambiente de produção: os padrões de utilização do sistema podem mudar (volume de dados, tráfego etc.) ou a infraestrutura do sistema pode ser modificada, porém, na maioria dos casos, a mudança mais frequente é a mudança de código. Empresas se tornam mais competitivas no mercado atraindo mais clientes com novas funcionalidades, alterando sistemas web, websites ou aplicativos móveis. As pipelines de Integração Contínua são fundamentais para "proteger" o ambiente de produção de possíveis regressões, então é muito comum que SREs mencionem pipelines de CI e testes durante discussões de *Post Mortem*.

Entrega Contínua (Continuous Delivery)

Uma vez que as mudanças são integradas com sucesso, o objetivo passa a ser a entrega da nova funcionalidade, ou a correção de um bug, para o cliente da maneira mais eficiente possível sem afetar usuários. Uma pipeline de Entrega Contínua (CD) envolve a criação de artefatos (também conhecidos como *builds*), aos quais são atribuídas versões, e o *deploy* desses artefatos para o ambiente de produção. Isso garante que a organização possa entregar novas funcionalidades aos usuários a qualquer momento através de um processo automatizado, rápido e confiável.

SREs devem colaborar com times de desenvolvimento para criar documentação relacionada a como reagir caso algum deploy apresente instabilidade (esse tipo de documentação é comumente encontrado em *playbooks*, assunto que será discutido no capítulo 11) e como um operador de um serviço deve interagir com a pipeline de Entrega Contínua. Essa documentação deve conter instruções para ajudar o operador a reverter a mudança aplicada recentemente e iniciar um novo deploy da versão anterior (versão estável). Apesar de essa medida ser conceitualmente simples e intuitiva para a mitigação de incidentes, a operação de fato talvez precise de instruções bem detalhadas para orientar o operador a utilizar alguma interface gráfica ou comandos em seu terminal. Uma vez que esse conhecimento é compartilhado, o próximo passo é, naturalmente, buscar a automação deste processo.

3.3 Testes e Quality Assurance

Testes e Quality Assurance (QA) realizam um papel superimportante em termos de Engenharia de Confiabilidade de Sites, um software bem testado ajuda a prevenir incidentes e evitar que bugs se manifestem no ambiente de produção.

Tipos de testes

Existem vários tipos de testes e essa terminologia deve fazer parte do vocabulário de um(a) SRE. Vamos conceituá-los:

- **Testes Unitários:** testes isolados que têm como alvo partes específicas do código de uma aplicação. Estes testes devem ser executados de maneira *offline* (sem interagir com outros serviços ou dependências remotamente).
- **Testes Positivos:** com uma determinada entrada (*input*) de dados, verificam se o software produz um resultado (output) esperado.

- **Testes Negativos:** testam se a aplicação trata um comportamento inesperado do usuário ou uma entrada de dados inválida graciosamente.
- **Testes de Integração:** testam a interação entre múltiplos serviços ou componentes.
- **Testes de Ponta a Ponta (End-To-End ou E2E):** executam uma "jornada do usuário" para testar funcionalidades críticas de um sistema. Um teste E2E pode se iniciar em uma tela de Login e executar uma série de passos programados para, por exemplo, adicionar um produto em um carrinho de compras e realizar uma compra para testar um website de e-commerce.

Não se deve confundir Testes Unitários com Testes de Integração: Testes Unitários *não* devem testar as funções de outros componentes, por exemplo, serviços de terceiros ou operações de um banco de dados, essas transações são candidatas a uma técnica chamada *"mocking"*. Quando um método ou função realiza chamadas para recursos externos à aplicação, o(a) autor(a) dos testes unitários deve criar respostas predeterminadas (*mocking*) para testar partes específicas da aplicação de maneira isolada. O foco dos Testes Unitários deve ser o conjunto de funções do serviço pelo qual o(a) autor(a) é responsável. Quando o(a) autor(a) dos testes decide que funcionalidades críticas específicas devem ser testadas com interações reais com outras entidades fora do contexto do serviço, deve-se implementar um Teste de Integração ou E2E.

Testes de Integração são utilizados para certificar que as funções do serviço estão operando da maneira esperada ao interagir com outros serviços internos ou serviços de terceiros. Quando Testes de Integração envolvem softwares de entidades externas cujo código e infraestrutura estão fora do controle de implementadores de tais testes, esses testes podem se tornar instáveis (também conhecidos como *flaky tests*), apresentando ocasionais falhas ao serem executados múltiplas vezes sem quaisquer mudanças de código (devido a alguma instabilidade com o serviço de terceiros). Uma estratégia comum para lidar com *flaky tests* é a configuração de

tentativas recorrentes (*retries*) para tentar fazer o teste passar em execuções subsequentes, em caso de algum problema passageiro.

Cobertura de testes

Ferramentas de testes devem calcular o número de linhas de código de uma determinada aplicação e medir quantas delas são executadas pelos testes unitários, fornecendo dados para ajudar a medir a "cobertura" de testes e informando que funções ou até mesmo que linhas não estão sendo testadas. Cada linha de código que não é sujeita a testes apresenta um risco em potencial para uma boa-noite de sono dos engenheiros que estão *on call*.

Aqui está o exemplo de um relatório de cobertura de código de uma aplicação escrita em Go (Golang), utilizando o comando `go test -cover`, que é uma ferramenta de linha de comando do próprio Go:

```
go test -cover ./... | tail -n3
ok      github.com/themarcelor/golang-testing/src/api/services    (cached)
coverage: 45.6% of statements
ok      github.com/themarcelor/golang-testing/src/api/utils     (cached)
coverage: 83.9% of statements
```

SREs podem utilizar as métricas de cobertura de testes como um critério para formalizar que um determinado serviço está "pronto para produção" (ou *Production Ready*), incentivando os times de desenvolvimento a investirem mais esforços relacionados a testes. SREs podem, por exemplo, sugerir que apenas serviços com a cobertura de testes acima de 70% serão incluídos no suporte de 24 horas por dia e 7 dias por semana (a revisão de requisitos para operar em Produção é um tópico que será discutido com mais detalhes no capítulo 11, quando discutirmos ferramentas de SRE).

A imagem a seguir mostra uma pirâmide de testes invertida para ilustrar como os bugs são "filtrados" por diferentes tipos de testes:

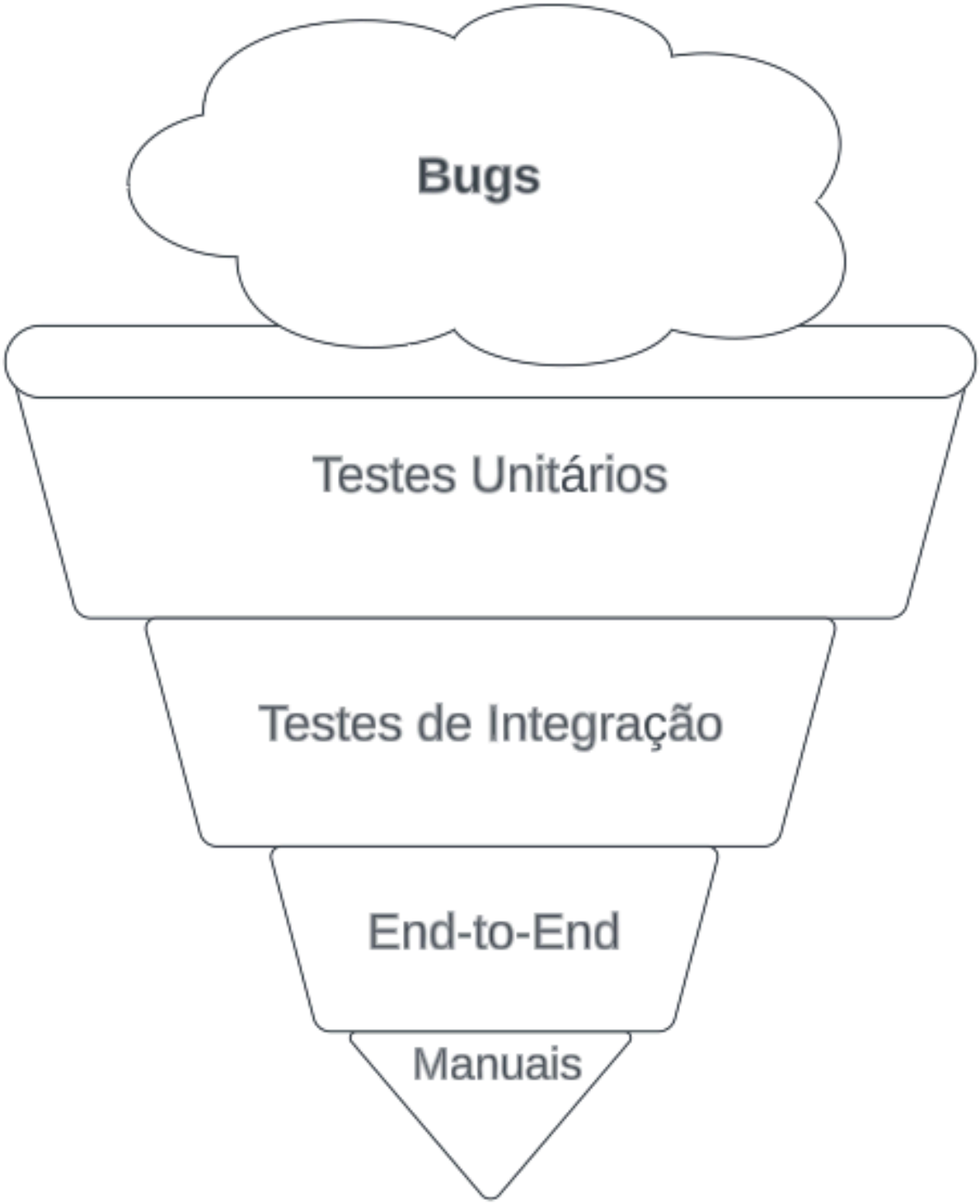

Figura 3.2: Pirâmide de testes invertida

Como podemos ver, a introdução de Testes Unitários deve ser priorizada devido ao ganho rápido de cobertura (*Test Coverage*), que

certifica que o código funciona como esperado, e ao baixo custo computacional, já que não necessitam de muitos recursos e infraestrutura para serem executados pois não possuem dependências externas. Um grande número de bugs deve ser capturado por Testes Unitários e, logo depois, as outras camadas seguem filtrando bugs através de Testes de Integração e testes E2E, até sua última camada, de "Testes Manuais", onde cenários extremamente específicos (ou que não podem ser automatizados) são verificados.

Outros tipos de testes

Outros testes que exigem mais recursos e infraestrutura são os testes de performance, que também possuem suas variações:

- **Testes de Carga (*Load Testing*):** testes que executam um alto volume de operações para verificar se a aplicação consegue acomodar uma carga de atividade que é esperada em condições normais diante de um sistema em produção.
- **Testes de Stress:** procuram determinar como a aplicação se comporta em situações de pico de atividade, com uma carga acima do normal (útil para analizar como o sistema se comporta durante eventos específicos, por exemplo, um site de vendas durante a Black Friday).
- **Testes de Estabilidade (*Soak Testing*):** similar ao Teste de Carga mas com uma execução duradoura, onde a aplicação processa transações por várias horas ou até mesmo dias para capturar anomalias que podem se manifestar apenas em situações onde a aplicação está em atividade por longos períodos (exemplo: vazamentos de memória ou outros eventos em que recursos são exauridos).

3.4 Estratégias de deployments para confiabilidade

Já que falamos sobre CI e os vários tipos de testes, vamos explorar também as opções de *deployment* que podem ser utilizadas em uma pipeline de CD:

- **Deployment por rotação (*Rolling Deployment*):** cada instância, réplica ou servidor, que abriga a versão atual da aplicação é substituída uma de cada vez. Neste tipo de *deployment*, a próxima instância só é substituída uma vez que a anterior inicia a nova versão da aplicação com sucesso.
- **Deployment Ativo/Passivo (*Blue/Green*):** nesta estratégia existem dois ambientes de produção, sendo um ambiente passivo, que não recebe requisições do usuário final, exposto apenas à rede interna para fins de acomodar atualizações, tarefas de manutenção e testes; e outro ambiente ativo, servindo requisições submetidas pelos clientes da aplicação. O *deployment* acontece no ambiente passivo para que as mudanças possam ser verificadas de forma segura sem afetar os usuários que continuam interagindo com o ambiente ativo e, logo depois, os ambientes trocam de papel: o ambiente passivo torna-se ativo e vice-versa (processo conhecido como *Swap*), fazendo com que a nova versão da aplicação fique exposta aos usuários. O processo se repete em *deployments* subsequentes. Se alguma coisa der errado, a troca é revertida para reativar o ambiente que já era estável.

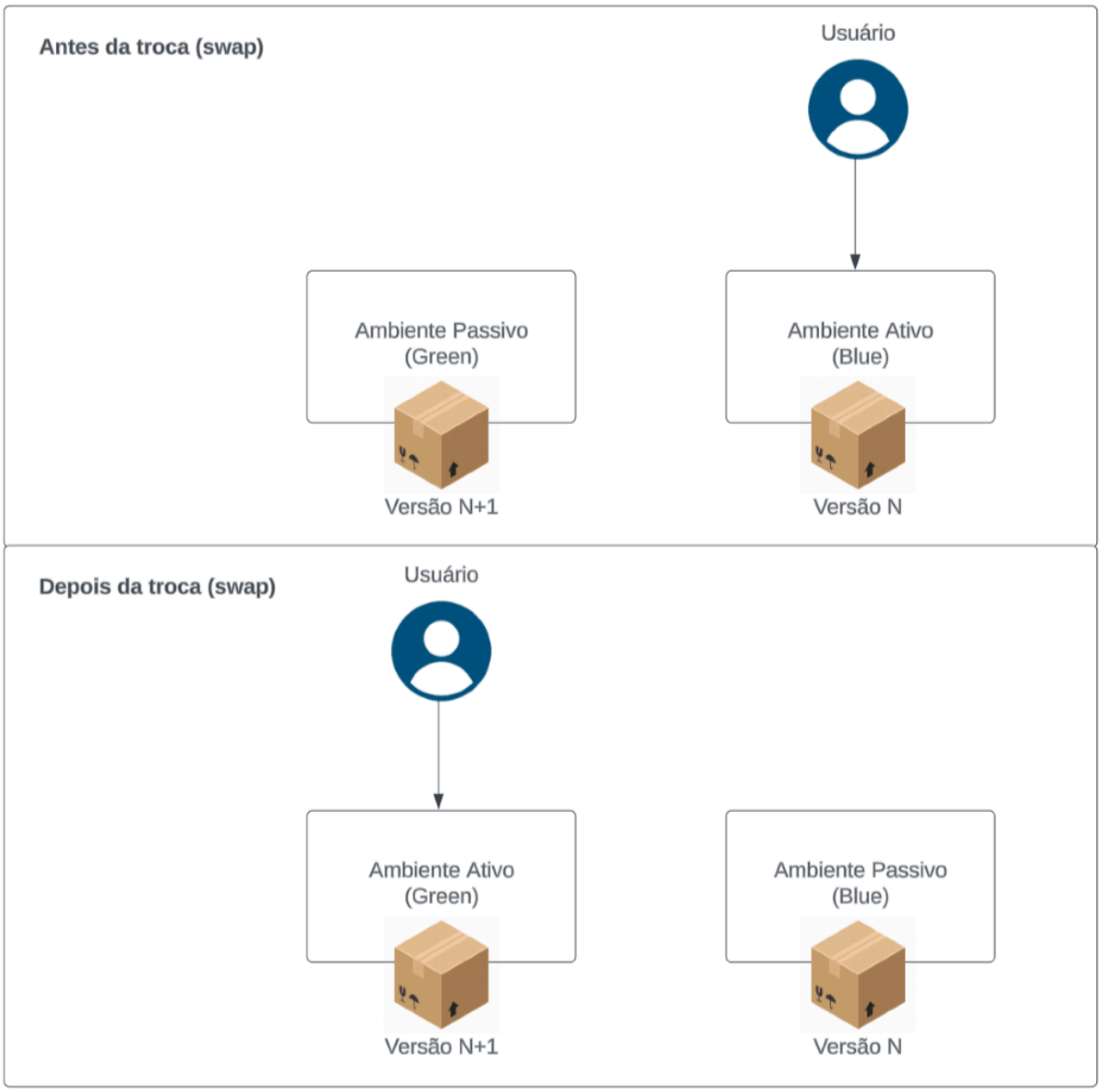

Figura 3.3: Blue/Green deployment

- **Canário (*Canary*):** há muito tempo, mineradores carregavam uma pequena gaiola contendo um canário enquanto exploravam minas de carvão para detectar vazamentos de gases nocivos. Como os pulmões do pequeno pássaro são sensíveis, os mineradores podiam determinar se era seguro avançar observando o pequeno animal. Seguindo o mesmo raciocínio, o *deployment* Canário envolve a entrega de novas versões do software realizada de maneira progressiva, geralmente expondo

as novas funcionalidades para um pequeno percentual de usuários e, uma vez que o sistema é observado e o time de engenharia confirma que ele ainda opera como esperado, aumenta-se o percentual de adoção da nova versão e o processo se repete até atingir 100%, concluindo o *deployment*.

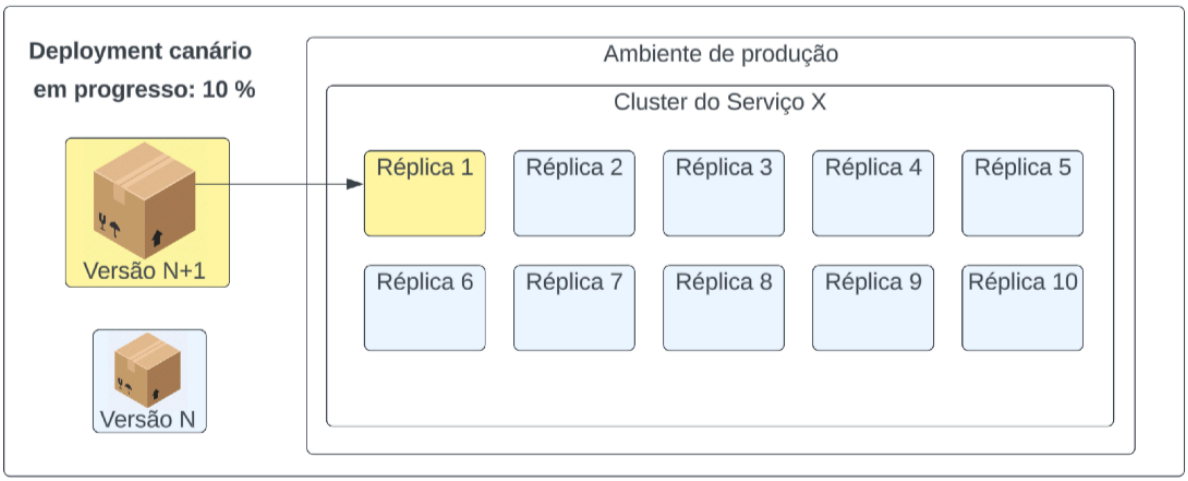

Figura 3.4: Canary deployment

Para ilustrar como uma boa estratégia de *deployment* é vital para a Engenharia de Confiabilidade de Sites, vamos imaginar um cenário em que um website de vendas de passagens aéreas está migrando para um novo sistema de pagamento. Digamos que o time de desenvolvimento se reuniu com SREs e decidiram que a migração para a nova funcionalidade seria introduzida através do *deployment* do tipo Canário. A migração para um novo sistema de pagamento exige mudanças de código em componentes críticos, portanto é muito arriscada, então implementa-se uma *release* progressiva em que, no primeiro estágio do *deployment*, o time de engenharia decide que apenas 1% dos usuários terão acesso à nova funcionalidade. E, se erros ou alta latência forem observados durante este exercício, a nova versão é revertida para restaurar a experiência desse pequeno número de usuários o mais rápido possível.

Para colocar esse exemplo em perspectiva: a Delta Airlines, uma companhia aérea dos Estados Unidos, teve um prejuízo de 150 milhões de dólares devido a uma indisponibilidade (*outage*) de 5 horas. SREs devem conhecer tais estratégias de *deployments* e colaborar com outros times de engenharia, principalmente aqueles responsáveis por *releases* e eventos de lançamento de novas funcionalidades (*go-live*), para discutir medidas que ajudem a conter o impacto de possíveis eventos problemáticos. Vamos explorar mais técnicas para melhorar a resiliência de serviços no capítulo 13.

3.5 Gerenciamento de releases

Como visto nas seções anteriores, SREs devem conhecer cada estágio do ciclo de vida do desenvolvimento de software para colaborar com outros times, discutindo e implementando melhorias. O processo de *release* de software é a parte relacionada à estratégia e a cadência em que as novas versões são enviadas para cada ambiente e como os times trabalham em conjunto com as mudanças de código.

Cada organização tem seu método de *release* alinhado com a natureza de seus sistemas e usuários. Nem todo negócio exige que novas versões de software sejam submetidas para produção imediatamente ou que o *deploy* seja coordenado para evitar indisponibilidade (*downtime*). Algumas empresas realizam novas releases uma vez por mês, outras, uma vez por semana. Porém, com o advento de DevOps, novas tecnologias e práticas estão surgindo para facilitar *deployments* e o gerenciamento de *releases*, possibilitando que sejam realizadas a qualquer momento.

Para ilustrar diferentes estratégias de *release* e analisarmos suas vantagens e desvantagens, vamos explorar 3 exemplos de *releases*:

1. Mensais

2. Semanais
3. Granulares e sob demanda

Releases mensais

Digamos que a empresa X oferece uma solução Web que recebe atualizações todo mês e cada nova versão compreende mudanças de código acumuladas durante o período de uma iteração de desenvolvimento. O ciclo de *release* é composto de 3 iterações que se sobrepõem periodicamente: Desenvolvimento, Estabilização e Finalização. Cada iteração possui sua própria *branch* no sistema de versionamento de código.

O diagrama a seguir mostra o fluxo desta estratégia e será nossa referência visual enquanto descrevemos cada passo da release número `60` (arbitrariamente definida para ilustrar o processo em andamento), que inclui as iterações de Desenvolvimento, nomeada como `D60`, Estabilização ou `S60` e Finalização `F60`.

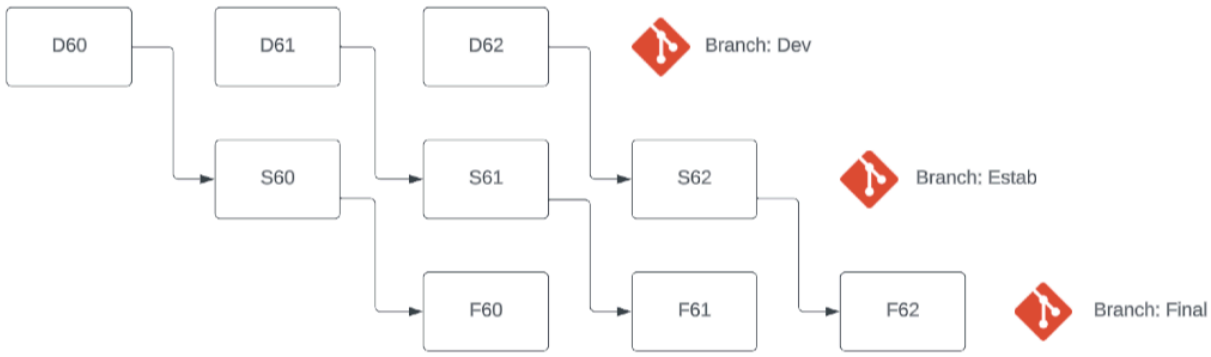

Figura 3.5: Fluxo de release mensal

Os times de desenvolvimento testam seu código localmente ou em *sandboxes* antes de submetê-lo para a *branch* de *Dev*. Um processo agendado atualiza o ambiente de testes através do *redeploy* de todos os serviços contendo as últimas mudanças de código e executa uma suíte de testes automatizados. Se todos os testes passarem, esse conjunto de mudanças, também conhecido como

build, é rotulado como estável (aplica-se uma tag no sistema de controle de versionamento).

Neste exemplo, os desenvolvedores da empresa X têm 2 semanas para trabalhar na iteração D60 e, no final da mesma iteração, o código referente à última *build* estável é promovido para a *branch* de Estabilização, o que marca o final da iteração D60 e o início da iteração S60, que compreende mais 2 semanas de testes adicionais (QA, Testes de Performance etc.). Em paralelo, desenvolvedores vão iniciar seu trabalho na próxima iteração D61. Quando os testes de performance forem concluídos, chega-se ao fim da iteração S60 e o código é, mais uma vez, promovido para a próxima *branch*: desta vez, a de Finalização (F60). Este é o estágio em que a *build* produzida pela *branch* de Finalização (F60) é finalmente submetida para o ambiente passivo de Produção para que possa ser verificada novamente antes do *Swap* (Lembre-se do *deployment* Ativo / Passivo ou Blue / Green descrito anteriormente).

Da mesma forma que S60 e D61 acontecem ao mesmo momento, as atividades de F60 acontecem em paralelo com a iteração S61, onde o codigo produzido na iteração D61 é testado (como visto no diagrama). Na empresa X, desenvolvedores não podem adicionar mudanças na *branch* de Finalização sem antes entrarem em uma reunião de "Gerenciamento de Mudança" para justificar suas intenções e explicar o impacto das alterações na *branch*.

Essa estratégia provê uma clara associação entre a versão que se encontra em uma determinada *branch*, seus respectivos ambientes e suas iterações, porém existe uma divisão entre os times da empresa X: Desenvolvedores trabalham em Dev, QA trabalham em Estabilização (com uma colaboração esporádica com desenvolvedores caso algum *bug fix* precise ser submetido para a *branch* de Estabilização) e o time de Operações foca nos artefatos produzidos pela *branch* de Finalização e burocratiza o acesso aos ambientes de Produção, formando-se "silos" de conhecimento entre os times e atritos no ciclo de vida do desenvolvimento.

Outra grande deficiência que foi deliberadamente colocada neste exemplo é a ausência de "Builds Herméticas". Na literatura religiosa-filosófica, Hermes Trismegistus é um alquimista que inventou um tubo de vidro que não poderia ser aberto, o que criou a expressão "hermeticamente selado", utilizada ao se referir a algo fechado de forma inviolável. Quando pacotes de software ou contêineres (*builds*) são reconstruídos baseados em alguma *branch* ou tag, a atualização de uma ou mais dependências pode acabar sendo absorvida por acidente e a *build* não será idêntica à que foi testada em outros ambientes em iterações anteriores, o que introduz um risco à *release*.

Portanto, uma *Build Hermética* é uma boa prática para aumentar a confiabilidade e a consistência de uma *release*. Os pacotes, binários ou contêineres que são testados em *Dev*, *Stage* ou *PreProd*, devem ser *exatamente* os mesmos encontrados na *release* que vai para Produção.

Releases semanais

Agora vamos analisar um caso mais dinâmico, que envolve *Releases* semanais: a empresa Y lança uma nova versão de seu SaaS (*Software as a Service*) para os clientes todo final de semana. O time de engenharia adota a nomenclatura "Bronze", "Prata" e "Ouro" para indicar onde encontrar uma determinada *build* em cada estágio do ciclo de *release*: Dev, QA e Prod, respectivamente.

Neste exemplo, o versionamento é baseado em uma contagem dos anos de existência do software e suas respectivas *Releases* semanais (`V<ano#>+<SEMANA>`), por exemplo: V10.51 é sucedido por V10.52, que, por sua vez, transiciona para V11.1 (adicionando 1 ao ano 10 já que ele completou o ciclo de 52 semanas).

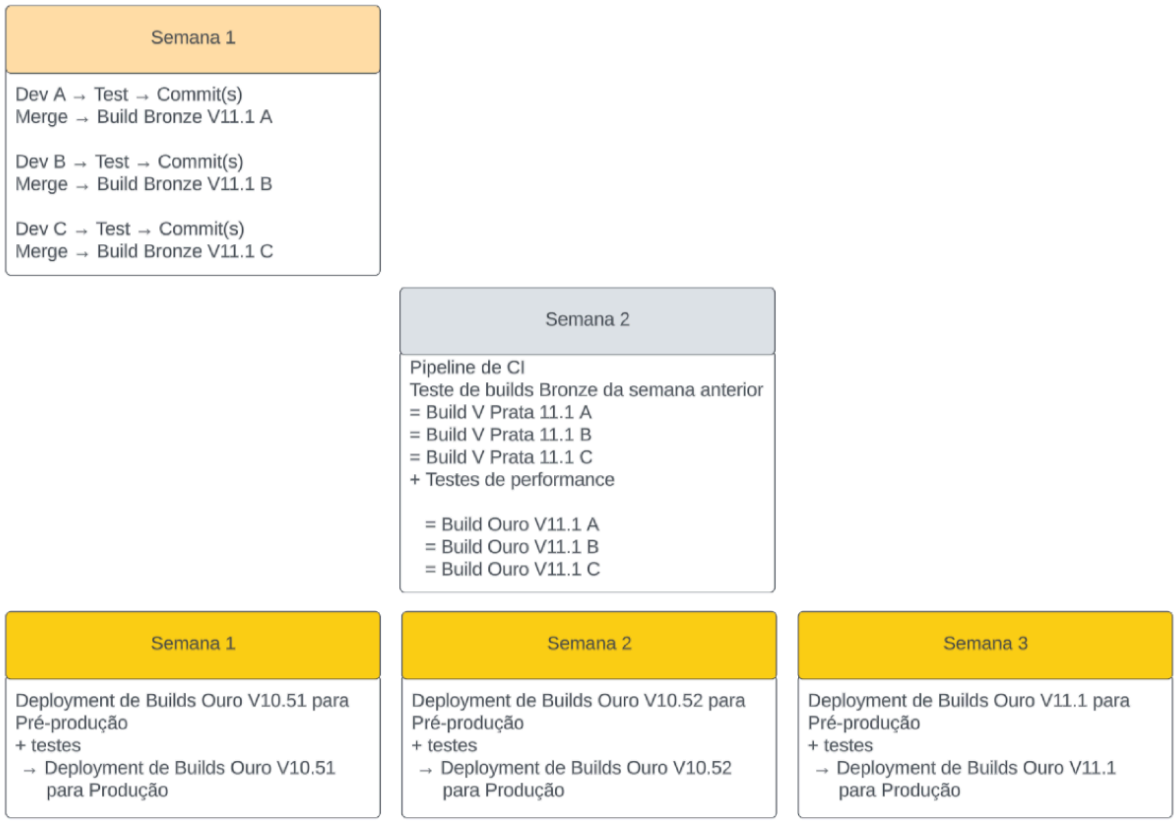

Figura 3.6: Fluxo de release semanal

Na semana 1, cada desenvolvedor testa as mudanças de suas *feature branches* em ambientes de *sandbox* antes de realizar um *merge* do código para a *branch* `develop`, que é a fonte das *builds* "Bronze" (ou Dev Builds). Os ambientes de *sandbox* compreendem uma versão simplificada da arquitetura do sistema com seus serviços e as respectivas *builds* "Prata", que são versões estáveis que já foram certificadas em ambientes de QA, então desenvolvedores sempre testam suas mudanças juntamente da última versão estável de outros serviços. No final da semana de desenvolvimento, todas as mudanças são agrupadas em *builds* "Bronze" para serem testadas nos ambientes de QA na semana seguinte.

Na semana 2, as imagens produzidas a partir da *branch* `develop` (*builds* "Bronze") são submetidas a um ambiente de Integração Contínua (*CI Environment*) onde todos os componentes do sistema são testados juntos. Se todos os testes passarem, a mesma imagem

"Bronze" recebe uma nova tag e é promovida à *build* "Prata" (que é a mesma *build* utilizada nos ambientes de *sandbox* mencionados anteriormente). Repare que o time de engenharia da empresa Y adota a prática de Builds Herméticas, já que suas pipelines de desenvolvimento constroem imagens Docker com a tag "Bronze" e essas mesmas imagens recebem diferentes tags ("Prata" e "Ouro") à medida que são promovidas para os próximos estágios do processo de *release* (imagens jamais são construídas fora do estágio de Dev). Ainda na segunda semana, durante o estágio de QA, as imagens Docker contendo uma tag "Prata" são submetidas aos ambientes de testes de performance onde são sujeitas a mais testes de carga. No final da semana 3, se todos os testes forem satisfatórios, a *build* "Ouro" é produzida.

Na semana 3, a *build* "Ouro" é instalada no ambiente de Pré-produção, onde a empresa Y permite que alguns clientes específicos tenham acesso exclusivo a novas funcionalidades que estarão disponíveis na próxima versão. Alguns testes adicionais são realizados em Pré-produção e, finalmente, no final da semana, a *build* "Ouro" tem seu *deploy* realizado em Produção. E assim é concluído o processo de *release* semanal.

Essa cadência de *releases* coloca a empresa Y em outro patamar de competitividade em relação à empresa X, pois entrega novas funcionalidades com uma frequência maior e tem um processo de gerenciamento de artefatos mais confiável. Porém, os múltiplos times de desenvolvimento ainda não têm a autonomia para realizar *deployments* em Produção a qualquer momento.

Releases granulares e sob demanda

Neste último exemplo vamos descrever o processo de *release* da empresa Z, que permite que qualquer time de desenvolvimento entregue novas versões de seus serviços a qualquer momento.

Basicamente, cada time de desenvolvimento da empresa Z tem sua própria pipeline para que engenheiros(as) possam escrever código,

enviar *commits* e realizar *merges* no repositório de código de seus serviços, e tem total controle sobre o *deployment* de suas mudanças para qualquer ambiente. Em seu estágio de desenvolvimento (Dev), cada mudança em um *Pull Request* é sujeita a testes que são executados em um ambiente de Integração Contínua. A mesma mudança é aplicada no ambiente de Stage e Pré-produção para Testes de Integração adicionais (testes de carga são opcionais, executados de acordo com natureza da nova mudança de código) e os próprios engenheiros podem executar a pipeline de Entrega Contínua e iniciar o *deploy* da mudança em Produção, ou seja, o time responsável pelo serviço gerencia sua própria *release*.

A estratégia de *deployment* Canário é utilizada para a *release* em Produção. Inicialmente, a versão N+1 tem seu *deploy* realizado e substitui a versão atual N em apenas um fragmento da frota de réplicas do serviço (por exemplo: instalando a versão N+1 em apenas 1 de 10 servidores que abrigam as instâncias do serviço). O sistema é observado para que se confirme que não há nenhuma regressão (*errors*, alta latência, impacto na integridade dos dados ou quaisquer outros *events* inesperados), caso o ambiente de produção permaneça saudável, mais réplicas são substituídas com a versão N+1.

A empresa Z possui uma arquitetura de alta disponibilidade (*high availability*) com múltiplos *data centers* na América Do Norte que abrigam clusters de serviços na costa Oeste e na costa Leste. Portanto, caso algum comportamento inesperado se manifeste no cluster de um serviço que está operando no *data center* da costa Leste, existem automações para redirecionar o tráfego dos usuários para o *data center* da costa Oeste, restaurando, desta forma, a experiência dos usuários rapidamente. Esta arquitetura pode ser visualizada da seguinte forma:

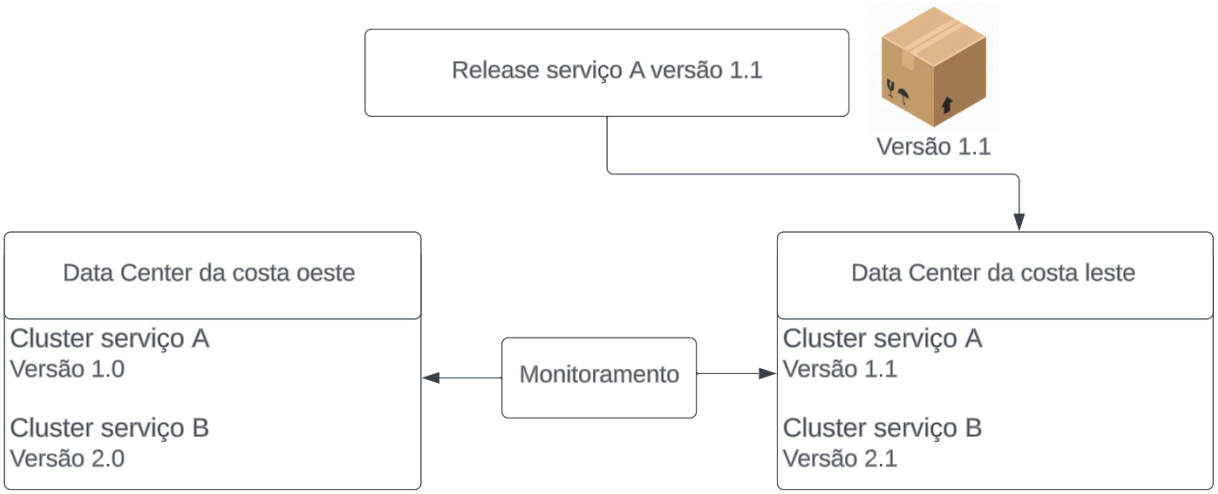

Figura 3.7: Fluxo de release granular

O redirecionamento de tráfego entre *data centers*, que é utilizado para uma ágil resolução de problemas que impactam usuários, é conhecido como **Site Failover** e será mais explorado no capítulo 13.

Este método de *release* permite que diferentes times realizem *deployments* granularmente e não fiquem presos a uma frequência específica como vimos nos exemplos anteriores. Porém, tal método só é viável com um investimento no design e implementação de serviços para que eles possam operar em harmonia com uma infraestrutura de alta disponibilidade que tem toda sua infraestrutura e bancos de dados replicados em múltiplas regiões.

3.6 O serviço está em Produção, e agora?

Neste capítulo nós discutimos ambientes, pipelines de CI/CD, testes, *deployments* e diferentes tipos de *release*. Para concluir, devemos ter em mente que, uma vez que uma nova versão do software aterissa em produção, o trabalho não acaba, pois é necessário que SREs continuem colaborando com o time de engenharia para

garantir que o software cumpra os requisitos operacionais e permaneça estável, mantendo os clientes felizes.

Assim como o software está em constante evolução, sujeito a mudanças, os processos de Engenharia de Software adotados em cada passo do ciclo de desenvolvimento também devem amadurecer e passar por suas próprias mutações de acordo com o aprendizado obtido durante incidentes, através de discussões internas ou em reuniões de *Post Mortems* coordenadas por SREs. A estabilidade de serviços em produção está alinhada com esse aprendizado contínuo, também conhecido como *feedback loop*, que é a expressão utilizada para descrever o ato de obter informação produzida na saída de um processo e utilizá-la na entrada do mesmo processo.

Em termos práticos: se um serviço produzir erros no ambiente de produção causados por uma mudança de código, SREs gerenciam o incidente e, com a assistência dos engenheiros(as) responsáveis pelo serviço (ou sua documentação técnica), executam o *deployment* de uma versão estável (aplicam uma mitigação) corrigindo o funcionamento do sistema. Desta forma, conclui-se que a pipeline deste serviço precisa de novos testes para obter uma melhor cobertura em relação à funcionalidade que apresentou falhas para os clientes. Então o time de desenvolvimento deve priorizar esta tarefa para prevenir que o mesmo problema ocorra novamente.

SREs buscam melhorias através do *feedback loop* para serviços que estão servindo clientes em Produção mas não devem se preocupar apenas em serem reativos ou, em outras palavras, não devem esperar que incidentes ocorram para revelar oportunidades de melhorias técnicas ou ajustes em processos de engenharia. SREs munidos de conhecimento do ciclo de vida do desenvolvimento de software e dos processos adotados na organização na qual trabalham devem ser proativos: devem coordenar simulações de incidentes (tema do capítulo 15) e continuamente certificar serviços em relação a seus requisitos operacionais.

Boa parte dos requisitos operacionais que são necessários para manter a confiabilidade de serviços em produção são cumpridos por uma boa arquitetura de software confiável, com uma infraestrutura e plataformas de *runtime* bem configuradas. Estes são conceitos que serão mais explorados no próximo capítulo.

Parte 2: Infraestrutura

Capítulo 4
Arquiteturas de software confiáveis

SREs devem possuir uma noção de arquitetura já que esse conhecimento permite a rápida identificação dos elementos que compõem um sistema e como eles interagem entre si, o que acelera o processo de triagem de um incidente em potencial. Tal conhecimento também ajuda a desenvolver a intuição para questionar pontos sensíveis de um projeto de desenvolvimento de software, assim como a habilidade de inferir os melhores caminhos para encontrar a causa raiz de um problema.

Este capítulo vai explorar alguns conceitos de infraestrutura e design de aplicativos associados com 3 requisitos não funcionais que são muito importantes para SREs: Escalabilidade (*Scalability*), Disponibilidade (*Availability*) e Resiliência (*Resiliency*). Outro requisito não funcional bastante alinhado com SRE é a Observabilidade, mas este será o foco do capítulo 6.

4.1 Escalabilidade

Escalabilidade é a habilidade de manter o serviço operacional mesmo com o aumento da carga de trabalho. Um serviço "escalável" mantém sua performance dentro dos critérios de qualidade à medida que o número de requisições seja incrementado, e essa performance é mantida através de um bom design de software e de uma infraestrutura que seja capaz de lidar com tais mudanças, podendo até operar de forma "elástica" (que se adapta automaticamente de acordo com a utilização de recursos de hardware como CPU ou memória).

Para falar de escalabilidade, temos que entender o que é um *"cluster"* de uma aplicação. O *cluster* é o conjunto de uma ou mais cópias de uma aplicação, também conhecidas como instâncias, que é utilizado para criar uma redundância. O conceito de redundância, quando se trata de Engenharia de Software, é a prática de disponibilizar múltiplas cópias de componentes críticos para evitar a interrupção de serviços ou algum impacto relacionado a dados. Então a aplicação que opera em um *cluster* possui várias instâncias instaladas em múltiplos servidores (ou em outro tipo de infraestrutura como máquinas virtuais ou contêineres).

Os dois tipos de escalonamento mais utilizados são o vertical e o horizontal. No escalonamento vertical, operadores(as) adicionam mais recursos para a máquina onde a aplicação está sendo executada: um processador mais potente, mais memória, mais espaço em disco ou até mesmo um disco com uma cota melhor de leitura e escrita (IOPS, ou *Input/Output Operations*). Por outro lado, no escalonamento horizontal, operadores(as) adicionam novas máquinas ao *cluster*, também conhecidas como nós (*nodes*). Uma aplicação com muitos nós em seu *cluster* (*cluster nodes*) está apta a distribuir seu trabalho computacional e servir um grande número de clientes.

Para ilustrar, a empresa X, anteriormente conhecida como "Twitter", de acordo com um de seus artigos publicados em seu Blog de Engenharia, processa 500 milhões de "tweets" por dia com centenas de milhares de servidores em seus *data centers* (https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale).

SREs podem mitigar incidentes causados por um aumento súbito em tráfego através deste conhecimento de escalabilidade, e devem estar cientes também de que, para cargas de trabalho que não são tão intensas, um *cluster* com muitos nós provisionados desnecessariamente apresenta um risco financeiro, pois a infraestrutura não está alinhada com o princípio da frugalidade (evitar consumir mais do que o necessário). Existem muitas

oportunidades para projetos de gerenciamento de capacidade, que envolvem a colaboração com o time responsável pelo serviço, engenheiros de testes (QA ou SDETs) e SREs, onde a infraestrutura e a configuração de recursos computacionais de um serviço podem ser ajustadas baseado em testes de carga e medições de comparação e contraste, conhecidas em inglês como *benchmarking*.

4.2 Disponibilidade

A disponibilidade garante que um serviço esteja sempre acessível. Em uma arquitetura confiável, o serviço deve possuir múltiplas instâncias da aplicação para evitar o temido *ponto único de falha* ou *Single Point of Failure* (SPOF). Por isso, é comum configurar um *cluster* para a aplicação e, como discutido na seção anterior, criar uma redundância.

Com a redundância, a aplicação serve requisições de múltiplas instâncias, mas qual instância recebe as requisições e, caso alguma instância falhe, como podemos prevenir que o cliente perceba a aplicação como indisponível? Quem controla tudo isso? Para realizar o roteamento de requisições e controlar o tráfego para instâncias saudáveis de um *cluster*, é necessário um balanceador de carga (em inglês, *load balancer*).

O balanceador de carga

O balanceador de carga é responsável por distribuir as requisições que viajam pela rede e têm como alvo múltiplas instâncias de um serviço. Essa distribuição de carga é acompanhada de verificações da saúde contínuas (*health checks*) que podem incluir ou excluir instâncias do serviço no fluxo do ingresso de requisições.

Isso quer dizer que, se uma instância responder com algum código de erro ou se o balanceador de carga não obtiver nenhuma resposta

dentro de um determinado limiar de tempo, os usuários não vão se comunicar com essa instância problemática pois ela será excluída do roteamento. Esse simples mecanismo aumenta significativamente a disponibilidade do sistema.

Existem dois tipos principais de balanceadores de carga: os de camada 4 (Transporte) e os de camada 7 (Aplicação). Essa terminologia vem do modelo de Interconexão de Sistemas Abertos (do inglês *Open Systems Interconnection* ou apenas OSI), que é um padrão ISO para comunicação de dados e define sete camadas, sendo que cada uma trabalha com protocolos específicos. Não vamos explorar as 7 camadas do modelo OSI neste livro, vamos apenas focar nestas duas camadas, pois estas são relevantes para distinguir o funcionamento dos balanceadores de carga.

Balanceadores de carga de camada 4 utilizam apenas informações de portas e protocolos para rotear pacotes e não inspecionam o conteúdo das mensagens, portanto são geralmente mais rápidos, porém limitados em termos de condições de roteamento. Um exemplo conhecido é o *Network Load Balancer*, ou apenas NLB, do AWS.

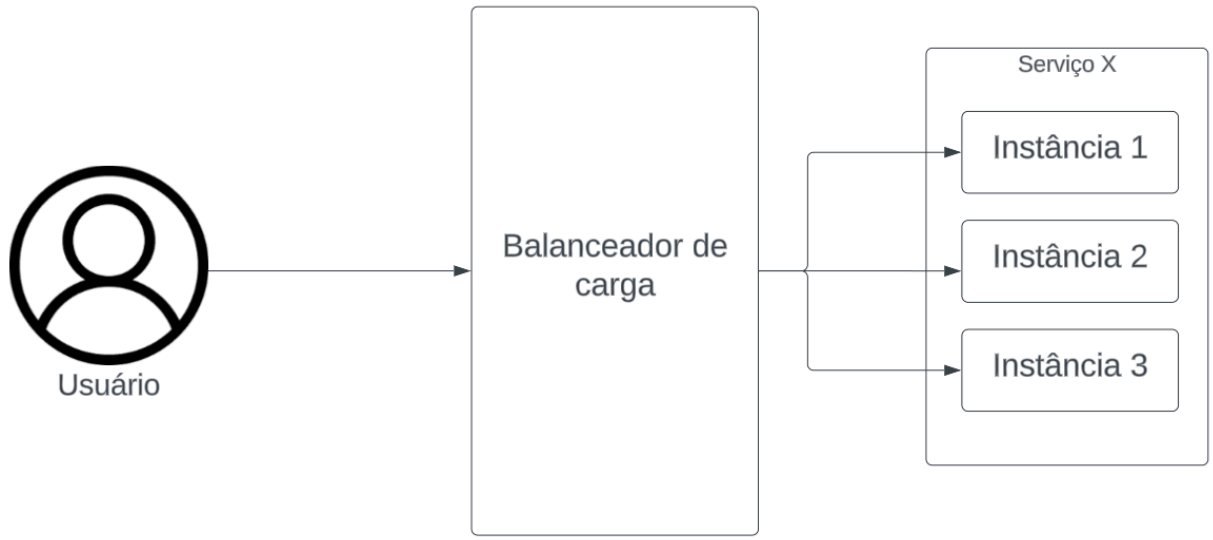

Figura 4.1: Balanceador de carga de camada 4

O outro tipo, o balanceador de carga de camada 7, trabalha com o conteúdo dos pacotes de requisições HTTP, portanto permite configurações de roteamento dinâmico baseadas em cabeçalhos HTTP (*headers*) ou caminhos de URL (*url paths*), o que os torna mais flexíveis pois operadores(as) podem, por exemplo, utilizá-lo para definir caminhos diferentes para serviços distintos ou, até mesmo, versões distintas.

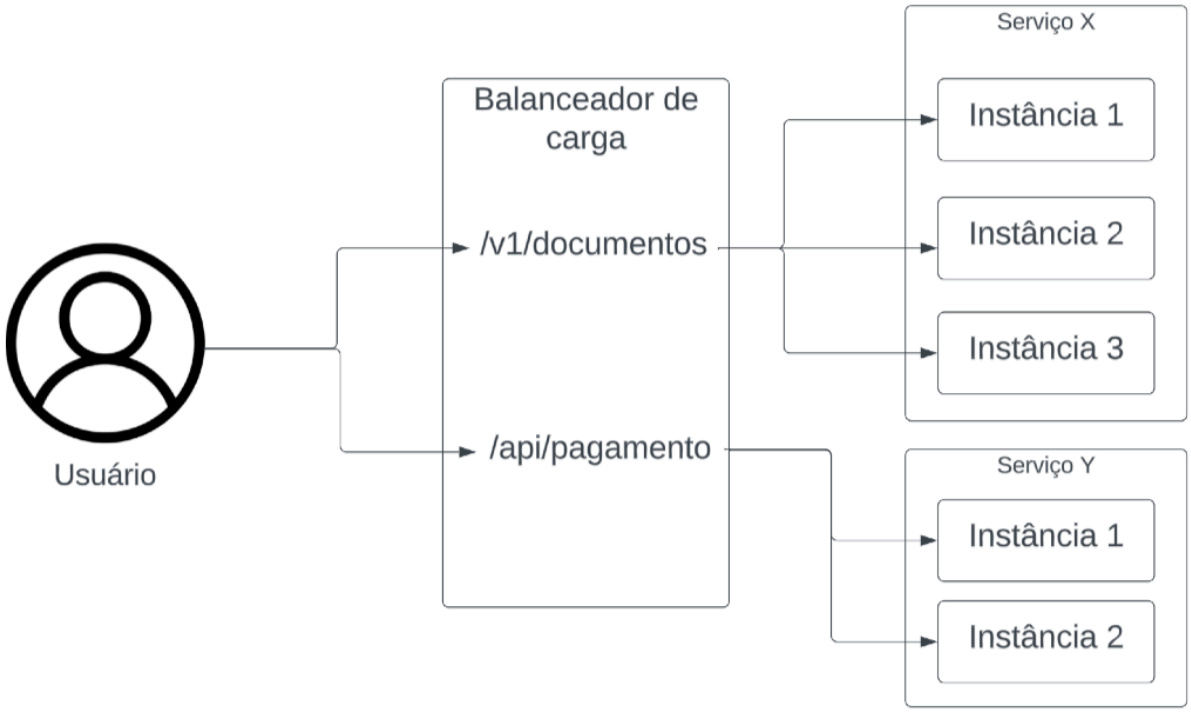

Figura 4.2: Balanceador de carga de camada 7

Os algoritmos mais comuns utilizados em balanceadores de carga são:

- **Revezamento de requisições (*Round Robin*):** encaminha as requisições selecionando cada servidor (instância do serviço) em sequência. A primeira requisição será destinada ao primeiro servidor, a segunda, ao segundo servidor, e assim por diante, até que envie requisições a todos os servidores e então volte para o começo da lista, sempre enviando requisições para cada

um. Este é o algoritmo padrão utilizado pela maioria dos balanceadores de carga.

- **Menos conexões (*Least Connection*):** determina o alvo do encaminhamento da próxima requisição medindo qual servidor entre as opções existentes tem o menor número de conexões. É uma opção que apresenta uma distribuição de carga mais inteligente mas, dependendo da natureza da aplicação, apenas o número de conexões não é suficiente para determinar a carga de trabalho de uma determinada instância, pois a atividade pode variar de acordo com o tamanho da requisição, com qual operação será realizada com tais dados etc.

No livro *The practice of cloud system administration: designing and operating large distributed systems* (Thomas A. Limoncelli, 2015), o autor compartilha uma história bastante interessante: durante o ataque de 11 de setembro de 2001, nos Estados Unidos, o website do grande canal de notícias CNN.com sofreu uma indisponibilidade causada pelo algoritmo de *Least Connection*. Devido ao atendado terrorista, muitas pessoas tentaram acessar o website ao mesmo tempo e isso, obviamente, sobrecarregou os servidores. À medida que servidores falharam e foram reiniciados, o servidor que completava o processo de inicialização primeiro era imediatamente enxurrado por requisições e sucumbia novamente, pois o algoritmo que buscava o alvo com o número mínimo de conexões estavam sempre roteando todas as requisições para aquele que estava mais folgado. A solução que os administradores de sistema utilizaram foi reiniciar todos os servidores ao mesmo tempo. Logo depois, todos concluíram o processo de inicialização aproximadamente ao mesmo tempo e o balanceador de carga finalmente dividiu as requisições de maneira mais justa e conseguiram, desta forma, restaurar o acesso ao website para que os usuários pudessem ler as notícias. Posteriormente, eles descobriram que existia um *upgrade* disponível para o software do balanceador de carga que introduzia uma funcionalidade de inicialização lenta (*slow start*), que prevenia que uma enxurrada de requisições fosse enviada para servidores que fossem adicionados recentemente.

Data centers, regiões e zonas de disponibilidade

Soluções na nuvem como o Amazon Web Services (AWS) e o Google Cloud Platform (GCP) possuem o conceito de regiões (*regions*) e zonas de disponibilidade (*availability zones* ou apenas AZs). Vamos explorar esses conceitos para entender como eles estão relacionados à disponibilidade, mas, primeiro, vamos falar sobre a *indisponibilidade*. A indisponibilidade pode ser causada por uma série de problemas: mudanças de código inseridas através do *deployment* de uma versão problemática, exaustão de recursos causadas pela ausência de escalabilidade, problemas de hardware etc. Nesta seção vamos focar mais na indisponibilidade causada por problemas relacionados à infraestrutura de *data centers*.

Uma máquina que abriga determinada cópia da aplicação pode ser impactada se um *data center* sofrer de alguma falha sistêmica causada por erro humano ou até mesmo um desastre natural como um terremoto ou uma inundação, por exemplo. Existe um universo de possíveis problemas que um servidor ou até mesmo a rede, ou qualquer outro componente de infraestrutura, podem enfrentar, e tais problemas podem impactar um ou mais nós de um *cluster*. Um fato curioso sobre problemas na rede elétrica, uma ameaça comum aos *data centers*, é que um número significante de ocorrências são causadas por esquilos viajando pelos cabos. O especialista em Cyber Segurança Cris Thomas até criou um site para mapear todas as ocorrências de interrupções de energia causadas por esquilos (https://cybersquirrel1.com).

Uma zona de disponibilidade, que será referenciada apenas como AZ de agora em diante, é basicamente um *data center* localizado em uma área geograficamente favorável. Uma região é composta por múltiplas AZs distribuídas estrategicamente em uma mesma área, as instalações de uma AZ podem ser localizadas no norte de uma cidade enquanto outra AZ existe no sul, permitindo que a região, como um todo, continue operando mesmo que alguma AZ enfrente algum problema. Quando se configura um *cluster*, as instâncias de

aplicações confiáveis são geralmente distribuídas em múltiplas AZs de uma determinada região para evitar a possibilidade de impacto relacionado aos problemas mencionados anteriormente (já que cada AZ é localizada em pontos geográficos diferentes). Além da distribuição de instâncias por AZs, operadores(as) podem também configurar aplicações de multirregião, onde múltiplos *clusters* podem existir em regiões diferentes, o que resulta em um custo adicional de infraestrutura, porém traz grandes vantagens em termos de confiabilidade.

O diagrama a seguir mostra duas regiões: `us-west-2` no estado de Oregon, nos Estados Unidos, e `us-east-1` ao norte do estado da Virgínia. Veja que nem toda região possui o mesmo número de AZs (e algumas regiões não suportam todos os serviços do catálogo do provedor de soluções na nuvem), então, em alguns casos de aplicações com dependências específicas, é recomendado planejar a configuração de infraestrutura do serviço e estudar que regiões são ideais para abrigar o *cluster*.

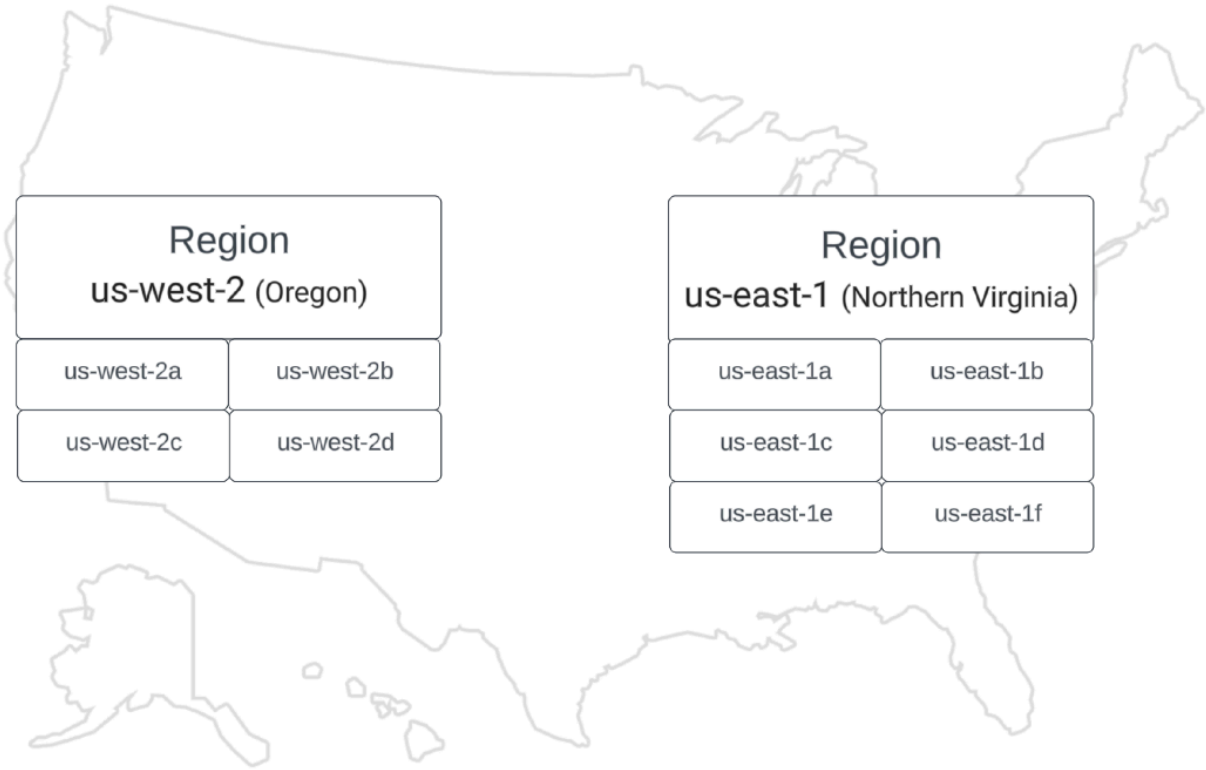

Figura 4.3: Regiões e Zonas de Disponibilidade

Uma arquitetura de alta disponibilidade deve levar todos esses conceitos em consideração e SREs devem colaborar com os times responsáveis pelos serviços para se certificar que os dados relacionados a infraestrutura do serviço estão devidamente catalogados, assim como a documentação técnica (*runbooks* ou *playbooks*), que deve ser facilmente localizada para auxiliar pessoas que estão *on call* a encontrar estes serviços e seus respectivos *clusters*. Afinal, durante o processo de triagem de algum incidente em potencial, SREs precisam investigar problemas na mesma região em que o serviço se encontra. Para saber mais sobre as regiões e AZs disponíveis no AWS ou GCP, mais informações podem ser encontradas em https://aws.amazon.com/about-aws/global-infrastructure/regions_az e https://cloud.google.com/compute/docs/regions-zones. Cada provedor de soluções na nuvem possui documentações com

orientações semelhantes para ajudar operadores(as) a configurar aplicações de alta disponibilidade.

Alguns times de operações implementam ferramentas para medir a disponibilidade de um serviço baseadas apenas em requisições simples que ficam continuamente "pingando" o endereço de uma determinada aplicação. Para websites de acesso externo, por exemplo, existem muitas soluções para monitoramento do tempo operante de um serviço, como Uptime (https://www.uptime.com) ou Site24x7 (https://www.site24x7.com). SREs, no entanto, observam o serviço além destes critérios, pois compreendem a diferença entre disponibilidade (*Availability*), tempo operante (*Uptime*) e confiabilidade (*Reliability*).

No livro *Implementing Service Level Objectives* (Hidalgo, 2020), o autor explica que todas estas palavras são frequentemente utilizadas para discutir a mesma coisa, porém possuem um significado diferente:

- **Tempo operante (*Uptime*)** é a resposta para a pergunta: O serviço está "de pé" e em execução?.
- **Disponibilidade (*Availability*)** já elabora a pergunta um pouco mais: O serviço está "de pé", em execução, e está respondendo à requisições? Está processando alguma entrada de dados (*input*) e produzindo algum resultado (output)?
- **Confiabilidade (*Reliability*)** foca em uma pergunta mais abrangente: O serviço está fazendo o que deveria fazer e está operando no nível de qualidade esperado?

Entre outros times de engenharia de software, essas palavras podem ser consideradas sinônimos, mas é uma distinção particularmente útil para SREs.

O teorema CAP

Este teorema declara que não é possível construir um sistema distribuído que possa garantir Consistência (C para *Consistency*), Disponibilidade (A para *Availability*) e Tolerância a partições (P de *Partition Tolerance*). É possível obter apenas uma ou duas opções, mas nunca as três simultaneamente.

- **Consistência:** quando o sistema é consistente, não interessa qual instância da aplicação está processando uma requisição, todas devem responder com a mesma informação. Essa característica é particularmente importante para sistemas que exigem que todos os clientes tenham acesso às informações mais atualizadas. Um exemplo clássico é o sistema bancário, onde usuários utilizam caixas eletrônicos em lugares diferentes e todos apresentam um saldo sincronizado. Para outros sistemas, como uma rede social, por exemplo, é possível operar satisfatoriamente com Consistência Eventual (*Eventual Consistency*), permitindo que clientes consultem o sistema e obtenham dados que possam estar desatualizados por algum tempo (enquanto os nós do *cluster* de armazenamento são sincronizados), já que não é necessariamente crítico que todos os usuários vejam as últimas postagens ou atualizações de algum determinado perfil imediatamente.

- **Disponibilidade:** o sistema está sempre disponível para responder às requisições. Cada requisição vai obter uma resposta independentemente do estado de cada nó do *cluster*. Se o *cluster* estiver parcialmente indisponível, os clientes ainda podem interagir com o sistema, mas não há garantia de que os dados obtidos estarão atualizados.

- **Tolerância a partições:** aqui, "partição" refere-se a uma falha de comunicação de rede, que se trata do cenário em que os nós do *cluster* não podem estabelecer conexões entre si, ou seja, não podem replicar atualizações de dados. Neste caso, o sistema tolera uma interrupção na comunicação entre os nós de seu *cluster*, permite que os clientes continuem interagindo com

o sistema, e pode sincronizar os dados depois que a conectividade for reestabelecida.

Vamos explorar as combinações analisando o que acontece quando uma partição ocorre: se o sistema priorizar Consistência, ele pode bloquear operações de escrita no banco de dados, tornando esta operação indisponível, e servir apenas operações de leitura até que a conectividade entre os nós seja reestabelecida. Se a Disponibilidade for priorizada, o sistema permite que novos dados sejam inseridos por usuários, mas nós diferentes vão armazenar informações distintas, criando uma inconsistência.

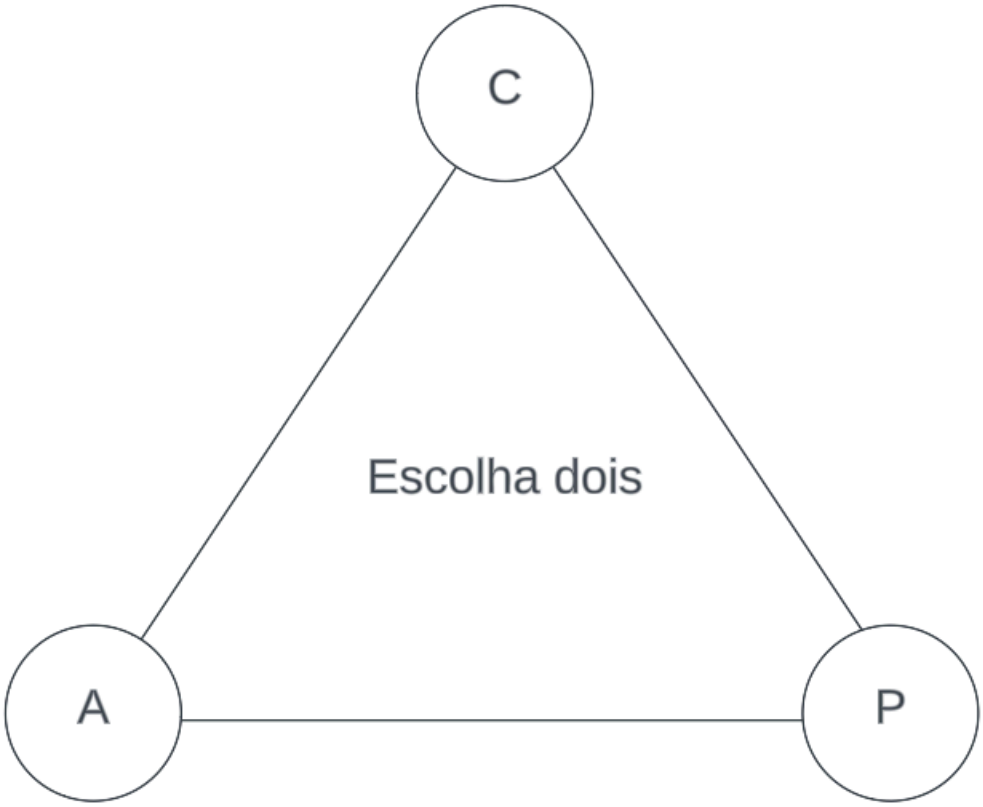

Figura 4.4: O teorema CAP

Este teorema é utilizado como uma referência para que engenheiros(as) de software possam refletir sobre suas arquiteturas e considerem comprometimentos ao escolher alguma funcionalidade e aceitar que decisões vão afetar outras características do sistema.

4.3 Resiliência

Um sistema é considerado resiliente quando consegue lidar com falhas graciosamente, detectando problemas e realizando uma recuperação eficiente. Para elaborar uma arquitetura resiliente, os fatores principais a serem considerados são:

- Redundância
- Infraestrutura como código
- Escalonamento automático
- Evitar falha em cascata
- Recuperação de desastres

Redundância

Uma arquitetura resiliente faz uso da Redundância, que também é uma característica da Disponibilidade, possibilitando que um serviço continue operacional mesmo que uma ou mais instâncias sofram de alguma falha. Como discutido anteriormente, a Redundância compreende o uso de múltiplas instâncias da aplicação, formando-se um *cluster* para evitar o *ponto único de falha*, e estas instâncias estão presentes em múltiplos *data centers*, distribuídas em múltiplas zonas de disponibilidade ou até mesmo em múltiplas regiões.

Infraestrutura como código

A infraestrutura como código (*Infrastructure as Code* ou apenas IaC) é uma prática bastante recomendada pois permite que operadores(as) provisionam infraestrutura programaticamente, criando redes, subnets, servidores, bancos de dados e quaisquer outras dependências de um sistema sem nenhuma atividade manual. Operações manuais consomem muito tempo e são suscetíveis a erros, o que torna a adoção de IaC bastante útil para a reprodutibilidade quando se configura a infraestrutura de um sistema.

Caso algum ambiente seja comprometido ou um ou mais componentes sejam acidentalmente deletados, a infraestrutura pode ser reconfigurada através de automação, o que será mais preciso e bem mais rápido do que recriar tudo manualmente. Existe também a vantagem de utilizar o código de provisionamento de infraestrutura como um recurso de compartilhamento de conhecimento, já que membros do time de engenharia podem revisar mudanças de código juntos e manter-se atualizados sob qualquer alteração.

Uma ferramenta de infraestrutura como código bastante utilizada na indústria é o Terraform (cuja alternativa de código aberto se chama OpenTofu), que possui uma linguagem própria para definir recursos que devem ser provisionados e gerencia o estado dos componentes de infraestrutura. O Terraform permite que operadores(as) executem planos através do comando `terraform plan` para observar o que será criado, modificado ou destruído em uma determinada infraestrutura e, logo depois, aplicar as mudanças com outro comando: `terraform apply`. Isso é possível através de inúmeros provedores (*providers*), que são componentes que interagem com APIs de vários outros serviços para criar, atualizar e deletar recursos. Existem provedores Terraform que facilitam o provisionamento de infraestrutura em todas as plataformas de nuvem mais famosas, o que ajuda bastante no gerenciamento de ambientes e no desenvolvimento de sistemas resilientes.

Escalonamento automático

Como discutido anteriormente na seção sobre Escalabilidade, se um sistema for bombardeado por um volume de tráfego fora do normal, deve ser resiliente e ajustar sua infraestrutura automaticamente para lidar com alterações na carga de trabalho. O serviço AWS EC2, por exemplo, possui uma funcionalidade chamada *Auto-Scaling Groups* (ASG), que monitora métricas e controla o provisionamento de novas máquinas EC2 de acordo com políticas de escalonamento (*scaling policies*), fazendo com que o *cluster* se torne elástico. Os ASGs monitoram a utilização de recursos de infraestrutura através de um

outro serviço AWS chamado *CloudWatch*, que disponibiliza métricas relacionadas a CPU, memória, atividade de rede, número de requisições no Balanceador de Carga etc., e, quando algum valor de quaisquer dessas métricas atinge algum limiar definido nas políticas de escalonamento, novas máquinas virtuais são criadas, realizando o escalonamento horizontal. Isso permite que um *cluster* aumente o número de nós automaticamente se, por exemplo, o serviço começar a receber um número maior de requisições e suas instâncias apresentarem um consumo maior de CPU e memória. Assim o *cluster* se adapta automaticamente a um aumento na carga de trabalho.

O Escalonamento Automático também é responsável por manter um número mínimo de instâncias e controlar o crescimento do *cluster* através da definição de um número máximo de instâncias. Se alguma instância falhar e o número de nós do *cluster* ficar abaixo do mínimo que foi configurado, uma nova instância deve ser provisionada automaticamente, o que permite que um sistema realize uma restauração automática e continue a servir requisições.

Evitar falha em cascata

A falha em cascata acontece quando algum problema ocorre em algum ponto do sistema e outros componentes que dependem dele, por suas vezes, apresentam falhas que impactam suas próprias cadeias de dependências, causando um efeito dominó. Por exemplo: se um serviço está lidando com uma grande quantidade de requisições e uma de suas instâncias é sobrecarregada, causando a queda de um dos nós do *cluster*, as outras instâncias do mesmo serviço terão que trabalhar com um *cluster* menor (o oposto de um escalonamento horizontal) e ainda processar o mesmo volume de requisições, o que pode causar a queda de outros nós do *cluster* ou aumento na latência do serviço, criando um problema sistêmico. O mesmo cenário se manifesta em situações em que um ou mais servidores enfrentam uma exaustão de recursos (total utilização de CPU ou memória, falta de espaço em disco etc.).

Existem várias técnicas para evitar a falha em cascata, vamos conhecer algumas:

- **Tempo limite de execução (*Timeouts*):** se uma instância está demorando muito para processar uma requisição, isso afeta a "vazão" de entrada e saída de dados de um serviço, impedindo outras requisições de serem processadas devido ao uso de recursos como CPU e memória. Se o serviço demora demais para executar uma consulta no banco de dados, por exemplo, talvez a instrução fornecida pelo cliente possa estar buscando uma quantidade massiva de dados, então o serviço pode interromper tal operação para se proteger da exaustão de recursos. Com a definição de um tempo limite de execução, o serviço abandona o processamento de atividades ineficientes para liberar recursos para continuar servindo requisições dividindo o tempo e recursos computacionais de forma mais justa.

- **Repetidor de requisições (*Retries*):** todo sistema está sujeito a falhas que podem ocorrer em um certo espaço de tempo, então o serviço que realiza requisições pode ser configurado para repetir tais chamadas para lidar com tais problemas ocasionais. Se o alvo da requisição enfrentar algum problema passageiro, por exemplo, uma falha de conectividade de rede associada a um servidor específico, em uma tentativa subsequente, o balanceador de carga vai direcionar a nova requisição para um outro nó do *cluster* onde, provavelmente, a transação poderá ser completada com sucesso. Ou, em caso de alguma transação de alta latência, que atinge o tempo limite de execução e faz com que um erro de *timeout* seja retornado para o cliente, quando o serviço realizar uma nova requisição, talvez o mesmo nó do *cluster* não esteja tão ocupado como estava a alguns milissegundos atrás. Múltiplas tentativas processadas automaticamente fornecem uma ótima experiência para o usuário pois os erros ocorrem por trás das cortinas sem que ele perceba. As tentativas, entretanto, devem ser implementadas

com algumas ressalvas. Configurar um número máximo de tentativas é necessário para evitar que uma transação inválida seja repetida levianamente. Outra boa prática é introduzir um tempo de espera entre as tentativas (conhecido como *backoff*), para que um grande número de tentativas não estresse o sistema acidentalmente e permita que o alvo tenha mais tempo para se recuperar. Implementações comuns incluem repetições com pausas exponenciais ou do tipo *fibonacci-backoff*, onde o serviço faz pausas de 1s, 2s, 3s, 5s, 8s, 13s etc.; entre as novas tentativas seguindo a sequência de Fibonacci.

- **Degradação de serviço ou resposta padrão (*Service Degradation or Fallback*):** se um serviço encontra-se em uma situação de stress, beirando a exaustão de recursos ou quando suas dependências estão operando com uma performance abaixo do normal, ele pode proativamente mudar seu comportamento e operar em um modo de baixa capacidade, produzindo respostas diferentes que não exigem o mesmo trabalho computacional. Imagine um serviço de streaming como YouTube ou Netflix. Se um ou mais componentes estiverem sob estresse, o sistema pode exibir o vídeo com uma qualidade mais baixa por um curto espaço de tempo, o que, às vezes, é imperceptível aos olhos do cliente. Em outras situações em que o serviço encontra-se sob pressão, o serviço pode retornar alguma resposta padrão para evitar o trabalho usual que seria executado, alguma informação predefinida ou a última resposta que foi processada que pode estar guardada em memória (*cache*).

- **Interrupção de requisições (*Circuit Breaker*):** esta técnica é baseada no dispositivo de segurança de instalações elétricas chamado "disjuntor", que é aquele que interrompe a corrente elétrica quando ela ultrapassa uma certa intensidade, prevenindo danos a outros componentes e até mesmo incêndios. Seguindo o mesmo raciocínio, para evitar que um serviço continue recebendo requisições quando se encontra em algum estado problemático, o cliente bloqueia o envio de novas

requisições baseado em um número de *errors* consecutivos. Dessa forma, o cliente não desperdiça recursos repetindo requisições inutilmente e o serviço pode tentar se recuperar sem o estresse adicional de continuar acomodando tráfego. O mecanismo de interrupção de requisições opera em três estados: inicialmente o circuito está fechado e existe comunicação entre o cliente e o serviço; logo depois, se o limiar de erros for alcançado, o cliente interrompe o fluxo de requisições destinado aquele determinado serviço, neste caso diz-se que o circuito está aberto. Depois de um tempo configurável, o circuito transiciona para um estado "semiaberto" onde o cliente envia apenas uma pequena quantidade de requisições para fins de teste e, se identificar falhas, abre-se o circuito novamente (interrompendo a comunicação) ou se a requisição for bem-sucedida, fecha-se o circuito e o sistema volta a operar normalmente.

- **Filas de mensagens mortas (*Dead Letter Queue* ou apenas DLQ):** as filas de mensageria armazenam mensagens que são publicadas por um cliente, também conhecido como *produtor* (*producer*), para que possam ser processadas por outros serviços que agem como *consumidores* (*consumers*) dessas mensagens. Quando a mensagem é processada com sucesso, ela é deletada da fila; por outro lado, se o processamento da mensagem falhar por qualquer razão (devido a algum problema com outra dependência no sistema, como erros ao tentar gravar informações no banco de dados, por exemplo), a mensagem continuará na fila para que consumidores possam tentar digeri-la novamente. Caso falhas consecutivas ocorram, a mensagem será direcionada para uma fila de mensagens mortas, o que é bastante útil para que as transações não sejam perdidas e o conteúdo de tais mensagens possa ser inspecionado para investigar problemas. Uma vez que o sistema estiver saudável novamente, os operadores(as) podem redirecionar as mensagens da fila de mensagens mortas para a fila principal e reiniciar o fluxo original para que as mensagens possam ser processadas pelos consumidores

novamente. O uso de filas de mensageria é bastante crítico para sistemas que devem tratar cada transação com muita importância para evitar problemas de consistência (se considerarmos um sistema bancário, uma transação perdida tem sérias consequências).

Recuperação de desastres

No primeiro capítulo do *Livro de SRE* (Murphy, 2016), encontra-se a seguinte citação: "Esperança não é uma estratégia". Esta frase é de autoria de Benjamin Treynor Sloss, fundador do termo "SRE" e aquele que deu origem ao primeiro time de Engenharia de Confiabilidade de Sites no Google. A frase nos diz que não podemos esperar que tudo fique bem e que desastres não ocorram. SREs devem formular planos para reagir a situações adversas e preparar mecanismos que auxiliem na restauração do sistema para um estado operacional e garantir a continuidade do negócio.

Planos de recuperação de desastres (*Disaster Recovery*, ou apenas DR) são elaborados de acordo com o impacto que a ausência ou a degradação de algum sistema pode causar ao negócio da organização. Um plano de DR é bastante importante pois, se os sistemas críticos de uma organização pararem de servir requisições ou interromperem seus trabalhos computacionais, a empresa pode sofrer um prejuízo financeiro severo.

Para ilustrar uma situação de desastre, vamos supor que um sistema de uma loja virtual tem sua infraestrutura provisionada em um *data center* na região X e que, devido a um incêndio no prédio que abriga os servidores, o website torna-se indisponível para seus usuários. Tal indisponibilidade pode acarretar em um grande número de vendas perdidas, usuários frustrados que vão procurar alguma loja concorrente, assim como o dano feito à marca e a reputação da empresa. Diante desta adversidade, a empresa deve recorrer a um plano de recuperação de desastres. SREs devem coordenar o incidente, reunir outros times de engenharia envolvidos no

procedimento e implementar os passos do plano de DR (que devem ser testados periodicamente).

Neste cenário envolvendo a loja virtual, vamos supor que o *data center* da região X está indisponível e seus técnicos não têm nenhuma estimativa de quando ele voltará ao seu nível de operação normal, então será necessário o provisionamento de uma nova infraestrutura em outro *data center* na região Y. O time de engenharia desta loja virtual utiliza tecnologias de infraestrutura como código (IaC), portanto executa as automações para reprovisionar as plataformas de *runtime* e suas dependências para que o sistema possa voltar a operar. Como, neste nosso exemplo, estamos ilustrando um bom plano de DR, vamos assumir também que os dados relacionados aos produtos, estoque, compras e pedidos foram replicados entre um banco de dados de "leitura e escrita" (primário) na região X e um banco de dados de "apenas leitura" (secundário) que sempre existiu na região Y. Então, uma vez que a recuperação do sistema for operacionalizada no data center da região Y, o banco secundário que serve de réplica pode ser promovido a primário e o sistema pode continuar inserindo e lendo dados, restaurando, desta forma, o acesso ao website para que os clientes possam navegar em seu catálogo novamente para realizar compras e verificar o status de seus pedidos.

Uma vez que o incidente for mitigado através da implementação do plano de DR, SREs devem conduzir uma reunião de *Post Mortem*, onde será feita a autópsia do incidente. Nesta reunião, deve-se mensurar o impacto da janela de indisponibilidade (com a ajuda de algum time de análise de dados, se for viável e necessário) para reportar uma projeção do número de usuários impactados, ou, de acordo com o nosso exemplo da loja virtual, o impacto financeiro baseado no volume de vendas que geralmente é observado no espaço de tempo em que o website estava fora do ar. Esta prática de *Post Mortem* envolve também a documentação da causa raiz do problema, para que todas as entidades interessadas possam discutir os eventos que culminaram no incidente, para fins de identificar

possíveis melhorias que possam prevenir sua recorrência ou acelerar o processo de identificação e restauração do sistema (o capítulo 13 contém mais informações sobre *Post Mortems*).

Existem dois parâmetros importantes que devem ser definidos quando se elabora um plano de DR: o Objetivo de Tempo de Recuperação, do inglês *Recovery Time Objective* (RTO), e o Objetivo de Ponto de Recuperação, do inglês *Recovery Point Objective* (RPO).

O RTO foca na janela de tempo entre o desastre e a recuperação, em quão rápido devemos transicionar entre a ocorrência de um desastre, sua identificação e a inicialização do processo de cura do sistema para dar continuidade ao negócio da empresa. Portanto, o RTO está relacionado aos serviços, às aplicações, websites e infraestrutura.

O RPO, por outro lado, foca no tempo entre o último *backup* de dados e o momento do desastre, definindo qual é a quantidade de perda de dados aceitável. Em outras palavras, o RPO define um ponto específico na linha do tempo em que, caso seja necessário retornar para tal ponto em caso de algum desastre no sistema, os dados encontrados sejam satisfatórios para que o sistema opere sem consequências severas. Portanto, o RPO foca em configurações de frequência de backups.

Para ilustrar esses conceitos utilizando a nossa loja virtual como referência, vamos dizer que SREs colaboraram com times de Gerência de Produto para definir um RTO de 4 horas. Este RTO foi baseado em projeções de impactos financeiros aceitáveis e no tempo necessário para executar as automações de recuperação de desastre, verificar o provisionamento da infraestrutura e testar o sistema para garantir o controle de qualidade. Se a empresa deseja um plano mais robusto para minimizar indisponibilidade para evitar o impacto financeiro em caso de desastres, deve investir em mais

infraestrutura ou otimizar o processo e a automação para acelerar a restauração do sistema.

Ainda utilizando os parâmetros do plano de DR da loja virtual como um exemplo, digamos que foi definido um RPO de 2 horas. Este RPO foi decidido baseado na frequência de *backups* do banco de dados: o time de engenharia da loja virtual configurou um backup completo (*full backup*) diário e um *backup* incremental (*incremental backup*) que ocorre a cada 2 horas. Portanto, caso algum problema de replicação entre a região X e Y venha a ocorrer e o banco de dados secundário na região Y perca a sincronia com o banco de dados primário que opera na região X, se o banco de dados primário for destruído ou se os dados forem corrompidos de alguma forma, será necessário recorrer aos *backups* para iniciar a restauração da última cópia de dados (do inglês *snapshot*) e existe uma janela de 2 horas entre esses *backups*. Isso significa que a janela de tempo em que dados foram perdidos não vai exceder as 2 horas determinadas pelo RPO.

Algumas empresas investem bastante em recuperação de desastres e até mesmo realizam simulações. Por exemplo, o Google organiza um evento anual chamado Teste de Desastre e Recuperação (*Disaster and Recovery Testing*, ou DiRT) onde catástrofes são simuladas ao desligar a energia de um *data center*, interromper todos os links de rede, ou até mesmo interrupções mais granulares de dependências críticas de infraestrutura como DNS (*Domain Name System*) ou LDAP (*Lightweight Directory Access Protocol*). Como Kripa Krishnan menciona em seu artigo: "Resistindo ao inesperado" (*Weathering the Unexpected*, ACM, 2012), eventos de testes de recuperação de desastres são veículos excelentes para encontrar problemas em sistemas e processos através de um ambiente controlado. Tais simulações devem testar não apenas a parte técnica do plano de DR, os scripts de provisionamento de infraestrutura e sistemas de *backup*, mas devem também ajudar a organização a ganhar mais confiança em seus processos, documentação e comunicação interna.

Agora que aprendemos o que torna uma arquitetura de software escalável, com alta disponibilidade, resiliente, e conhecemos componentes de infraestrutura que facilitam o cumprimento desses requisitos não funcionais, no próximo capítulo vamos falar sobre a camada onde as aplicações são executadas: as plataformas de *runtime*.

CAPÍTULO 5
Plataformas de runtime

SREs devem familiarizar-se com as plataformas de tempo de execução (*runtime*) utilizadas em sua organização, pois elas representam uma parte da infraestrutura que é crítica para um sistema confiável. As plataformas de *runtime* são sub-sistemas que gerenciam a execução de uma determinada aplicação e suas múltiplas instâncias, fornecendo interfaces para que operadores(as) possam manipular configurações e obter telemetria dos processos em atividade.

Através do processo de construção (*build*), a aplicação tem seu código preparado para *deployment*, seja através de compilação seguida da geração de um arquivo binário ou apenas um empacotamento para facilitar a transferência de arquivos. Uma vez que a *build* estiver pronta para *deployment*, o operador utiliza a plataforma de *runtime* para inicializar a aplicação.

Times que buscam confiabilidade exploram as vantagens de uma plataforma de *runtime*, que oferecem alta disponibilidade através do gerenciamento de *clusters* de aplicações e ajudam serviços a operarem de maneira mais resiliente, facilitando a configuração de várias instâncias de um serviço e orquestrando o *deployment* de suas versões em vários servidores ou máquinas virtuais ou até mesmo contêineres (ou *pods* do Kubernetes).

As plataformas de *runtime* também se responsabilizam por instalar e configurar o ambiente de execução (*runtime environment*), que são os softwares que executam o código de uma aplicação. Por exemplo, uma aplicação Web desenvolvida com a linguagem Java precisa de um servidor Web como o Apache Tomcat ou o JBoss da Red Hat. Uma aplicação Python precisa apenas de uma máquina que possua o Python instalado para processar as instruções de um ou mais scripts. Esta é mais uma responsabilidade que fica nas mãos da plataforma

de *runtime*, permitindo que operadores(as) abstraiam a complexidade da configuração dos ambientes que executam a aplicação.

Tais abstrações são bastante úteis para que um time de desenvolvimento ganhe mais velocidade e preocupe-se apenas em realizar o *deployment* e observar sua aplicação em atividade, sem se preocupar com como as instâncias são distribuídas ou como o ambiente de execução está configurado. Algumas organizações são compostas de times de engenharia que têm a liberdade para implementar funcionalidades em qualquer tecnologia de sua escolha, ou mesmo realizar experimentos com novidades da indústria, componentes de infraestrutura diferentes e novas linguagens de programação. Tais experimentos são facilitados pela plataforma de *runtime*.

Sob o ponto de vista de SREs, a missão é gerenciar a confiabilidade de múltiplos serviços e, já que os mesmos podem operar com diferentes tecnologias ou linguagens de programação, SREs precisam ser *generalistas* para prover suporte e participar de discussões com um mínimo de conhecimento operacional sobre os componentes críticos e a infraestrutura que se encontram no ambiente de produção. Porém, ocasionalmente, SREs devem estar preparados(as) para se aprofundar nos detalhes técnicos de baixo nível e conduzir investigações ao lidar com incidentes ou ao discutir a causa raiz de um problema durante uma reunião de *Post Mortem*. Em alguns casos, para se aprofundar em tais investigações ou para aplicar mitigações aos incidentes, SREs precisam focar nas plataformas de *runtime* nas quais os serviços são executados.

5.1 Soluções On-Premises

Se uma organização faz o *deployment* de aplicações em seu próprio *data center*, diz-se que tal aplicação que opera em uma

infraestrutura privada é conhecida como "on-prem" (*on-premises*). Neste modelo, os administradores de sistemas devem configurar cada servidor para acomodar as aplicações de acordo com suas tecnologias: ambientes de execução correspondentes à linguagem de programação, dependências etc.; e devem utilizar alguma solução para orquestrar o *deployment* das múltiplas instâncias da aplicação em cada um dos servidores.

Para *data centers* privados, existem soluções proprietárias como o Tanzu Application Platform, da VMWare, ou Open Source como Docker Swarm ou Kubernetes, que são utilizadas para facilitar a distribuição de aplicações e contêineres através de vários servidores. Os administradores de sistemas da organização podem também criar suas próprias ferramentas para gerenciar *clusters* de aplicações e a infraestrutura do *data center*.

Um *data center* privado oferece mais autonomia em termos de Engenharia de Confiabilidade, já que SREs podem obter telemetria em todas as camadas, desde *Top-of-rack* (ToR) *switches* (dispositivos de redes utilizados para conexão múltiplos servidores) até métricas emitidas pelos vários componentes de baixo nível em um *data center*, e até mesmo colaborar com técnicos de *data center* para introduzir mais ferramentas monitoramento, automatizar soluções para problemas recorrentes ou explorar mais opções para gerenciar gastos de infraestrutura.

Big Techs como Meta, Google ou X possuem seus próprios *data centers* e seus engenheiros desenvolveram ferramentas para gerenciar seus serviços e suas respectivas configurações. Essas organizações investem bastante em sua infraestrutura para manter essa autonomia, o que não é viável ou mesmo rentável para organizações cujo sistema possui uma proporção menor de usuários e serviços.

5.2 Soluções na Nuvem

Com o advento de plataformas de *runtime* oferecidas como serviço, times de desenvolvimento de software utilizam tais soluções de computação em nuvem para focar cada vez mais em requisitos funcionais. Isso acelera a entrega de novas funcionalidades para seus clientes, enquanto a infraestrutura é abstraída, aliviando operadores(as) da responsabilidade de provisionamento de hardware (servidores, discos, equipamento de redes etc.), mas oferecendo interfaces para gerenciamento e configuração. Vários provedores de soluções na nuvem como Google Cloud, Microsoft Azure ou Amazon Web Services fornecem plataformas de *runtime* e outros serviços que aliviam a carga de trabalho de manutenção de infraestrutura significativamente.

Provedores de soluções na nuvem publicam o SLA (*Service Level Agreement*) de cada um de seus serviços. O SLA (ou Acordo de Nível de Serviço) descreve as diretrizes da confiabilidade oferecida por um determinado serviço, por exemplo, os serviços de IaaS (*Infrastructure-as-a-Service*) de plataformas da nuvem como Amazon Web Services (AWS) e Google Cloud Platform (GCP). O Amazon Elastic Compute Cloud, também conhecido como EC2, garante um *uptime* de quatro noves, o que significa que, se o cliente deste serviço identificar uma indisponibilidade em uma determinada região e as evidências (*logs*) comprovarem que o EC2 estava operando entre 99.99% e 99.0% de *uptime*, um crédito de 10% será aplicado na conta mensal e se a degradação do serviço for mais severa, o crédito pode variar entre 30% e 100%. Já o Google Compute Engine (GCE) garante um *uptime* de três noves (99.9% de SLA) no nível padrão (Standard tier) e tem créditos similares se o serviço operar abaixo de dois noves (mais informações em https://aws.amazon.com/compute/sla/ e https://cloud.google.com/compute/sla). O SLA é um fator importante a ser considerado quando uma organização está avaliando a contratação de serviços na nuvem, vamos nos aprofundar mais nos conceitos de SLA, SLO e SLIs no capítulo 9.

Outra vantagem da adoção de PaaS em termos de SRE é que as plataformas de *runtime* e os serviços na nuvem estão em conformidade com os requisitos operacionais que facilitam a jornada para a melhoria da confiabilidade de sistemas. No livro *Practice of Cloud System Administration* (Limoncelli Thomas A, et. al., 2014), o autor lista vários requisitos operacionais:

- Configuração (*Configuration*): o serviço deve ser configurado por meios automatizados.
- Inicialização e Interrupção (*Startup and Shutdown*): o serviço deve ser iniciado quando a máquina (servidor, VM, contêiner etc.) que abriga sua instância entrar em operação. Se o sistema operacional receber um sinal de *shutdown*, o serviço deve encerrar suas atividades graciosamente.
- Fila de drenagem (*Queue Draining*): caso necessário, o serviço deve possuir um mecanismo para drenagem onde ele para de aceitar novas requisições, mas continua trabalhando naquelas que já estão em processamento.
- *Software upgrades*: deve ser possível realizar *upgrades* (*releases*) sem indisponibilidade. Espera-se que o serviço seja atualizado através de substituição de suas réplicas à medida que são atualizadas.
- Backups e restaurações (*Backups and Restores*): deve ser possível realizar *backups* e restaurá-los enquanto o serviço está em operação.
- Redundância (*Redundancy*): muitas técnicas de confiabilidade e escalabilidade têm como requisito a habilidade de operar com múltiplas réplicas. Portanto, o design do serviço deve acomodar essa configuração.
- Bancos de dados replicados (*Replicated Databases*): serviços que acessam bancos de dados devem fazê-lo de maneira que suportem escalabilidade do *cluster* da base dados, utilizando réplicas configuradas como "somente leitura" (*read-only replicas*).
- Troca em funcionamento (*Hot Swaps*): componentes do serviço podem ser trocados sem causar um incidente de completa

indisponibilidade do serviço.

- Interruptores de funcionalidades (*Feature Toggles*): um parâmetro de configuração deve estar presente para habilitar ou desabilitar uma funcionalidade do serviço (vamos explorá-los no capítulo 13).
- Degradação graciosa (*Graceful Degradation*): o serviço age de maneira diferente se estiver sobrecarregado ou se outros serviços (dependências) estão indisponíveis.
- Controle de acesso e limites (*Access Control and Rate Limits*): se o serviço fornece uma API, esta deve possuir uma Lista de Controle de Acesso (*Access Control List* ou apenas ACL) que determina quais usuários têm permissão (ou quais têm seu acesso negado), e uma configuração para limites baseados na frequência de requisições.
- Controles de importação de dados (*Data Import Controls*): se um serviço importa dados periodicamente, devem ser estabelecidos mecanismos que permitam que operadores(as) controlem quais dados são aceitos, rejeitados ou substituídos.
- Monitoramento (*Monitoring*): fornecer manutenção a um serviço em produção requer visibilidade em como o serviço está operando (observabilidade será explorada nos capítulos 6, 7 e 8).
- Auditoria (*Auditing*): em algumas organizações, é necessário que o sistema produza *logs* de acesso e mudanças na configuração de permissões de um serviço para que ele seja examinado, e aprovado, em auditorias de segurança.
- Coleta de exceções (*Exception Collection*): quando o serviço gera uma exceção, ela deve ser coletada centralmente para análise.
- Instrumentação para Debug (*Debug Instrumentation*): o serviço deve gerar *logs* que são úteis para a investigação de problemas (*debugging*).
- Documentação para Operações (*Documentation for Operations*): desenvolvedores e engenheiros de Operações devem trabalhar juntos para criar um *playbook* com procedimentos operacionais para o serviço.

Nem toda solução de PaaS cumpre todos os requisitos listados acima, mas as soluções existentes no mercado atendem à maioria desses itens, especialmente quando se trata de configuração, redundância, monitoramento, instrumentação para *debug* e a documentação técnica que é essencial para operadores de tais plataformas.

Existem vários tipos de plataformas de *runtime* oferecidas por provedores de serviços da nuvem, e cada uma delas possui sua própria arquitetura e funcionalidades que permitem que operadores conectem outros serviços de acordo com o objetivo do projeto. Vamos explorar as seguintes opções e entender como elas funcionam: PaaS (*Platform-as-a-Service*), FaaS (*Function-as-a-Service*) e CaaS (*Container-as-a-Service*).

PaaS (Platform-as-a-Service)

Serviços da nuvem do tipo PaaS são plataformas de *runtime* que facilitam o *deploy* de uma determinada versão da aplicação em um *cluster*, geralmente empacotadas de acordo com a tecnologia ou linguagem de programação escolhida pelos operadores. Por exemplo, o Elastic Beanstalk (EB) da AWS ou o Google Application Engine (GAE) da GCP são opções de PaaS bem conhecidas na indústria. Ambos AWS EB e GCP GAE suportam várias opções de *runtime* (Python, Java, NodeJS, Golang etc.) e permitem o *deployment* de aplicações conteinerizadas (aplicativos empacotados através de imagens Docker). Essas soluções distribuem múltiplas instâncias de uma determinada aplicação em uma frota de servidores (máquinas virtuais) e oferecem escalonamento automático (*auto-scaling*) baseado na utilização de recursos (por exemplo: CPU, tráfego ou outras métricas) acompanhado de uma configuração de um número mínimo e máximo de instâncias.

O *cluster* de uma aplicação é composto de múltiplas máquinas virtuais (por exemplo: VMs do tipo AWS EC2 ou GCP GCE) e seu tamanho é definido de acordo com as configurações de Escalonamento Automático (*auto-scaling*). A plataforma monitora

cada instância e verifica se a aplicação está "saudável" através de *health checks* que são executados periodicamente (a cada 10s para o AWS EB e 30s para o GCP GAE, mas essas configurações podem ser ajustadas). Em caso de alguma falha, se uma instância da aplicação se tornar indisponível ou o *health check* obtiver uma resposta irregular, a plataforma vai provisionar mais uma VM e iniciar um novo *deployment* da aplicação para substituir a instância que está falhando e garantir o número mínimo de instâncias de acordo com o contrato de escalabilidade definido para este *cluster*. Esse cenário de "restauração automática" é bastante útil para SRE pois garante que o *cluster* continue operacional e seja resiliente em situações adversas, mesmo devido a problemas intermitentes como uma falha de hardware em algum servidor ou algum problema temporário em outras dependências (outros serviços) que interrompeu a máquina virtual ou aquela instância da aplicação.

Esse modelo de PaaS baseado em VMs possui uma arquitetura simples e a telemetria oferecida pelas plataformas de nuvem ajudam SREs a observarem a performance de cada serviço, assim como seus respectivos *clusters* e outras dependências de infraestrutura (componentes de gerenciamento de tráfego, armazenamento, redes etc.). Porém, uma desvantagem é que o provisionamento de VMs é uma operação que consome muito tempo, então o *deployment* de uma nova versão da aplicação que contém uma mudança para corrigir um bug crítico ou até mesmo a inicialização de novas VMs ativada pela escalabilidade automática para conter um aumento de tráfego são tarefas que podem levar minutos para serem concluídas, e o tempo é um fator vital quando os usuários de um determinado serviço estão frustrados e SREs precisam aplicar alguma mitigação para um incidente.

FaaS (Function-as-a-Service)

As plataformas de FaaS são baseadas em computação efêmera ou programas que são iniciados apenas para responder a eventos e que não precisam de provisionamento de servidores. Como a

infraestrutura já está pronta para executar funções, o operador só precisa empacotar o código, realizar o *deploy* e definir quais os eventos que devem iniciar a função (Requisições HTTP ou mensagens em uma fila de mensageria, por exemplo). Tais serviços são conhecidos como *"serverless"* (sem servidor) pois o operador pode abstrair todos os detalhes de baixo nível em relação à infraestrutura e focar apenas em sua função. AWS Lambda e GCP Cloud Functions são exemplos comuns de FaaS.

Serviços que operam em plataformas de FaaS são orientados a eventos e trabalham com tráfego que pode oscilar de maneira imprevisível, portanto são uma ótima solução em termos de escalabilidade. As funções são executadas para cada novo evento através da inicialização de uma micromáquina virtual baseada em kernel (*Kernel Virtual Machine*, ou apenas KVM), que é um processo mais eficiente comparado ao PaaS, mas que possui suas próprias limitações relacionadas ao tamanho das requisições, ao tempo de execução e concorrência (número de instâncias da mesma função que são executadas em paralelo).

SREs podem enfrentar situações problemáticas em que o número de invocações ou a duração de um lambda aumentam significativamente, fazendo com que um grande número de instâncias da mesma função sejam executadas em um curto espaço de tempo, causando erros devido ao estrangulamento de funções (*throttling*) uma vez que o limite de concorrência é atingido. Existem configurações para reservar um número de execuções concorrentes por lambda, para que elas possam ser priorizadas (*reserved concurrency*), e os gatilhos (*triggers*) de eventos que fazem a função acordar também podem ser ajustados para diminuir a vazão (*throughput*) de transações e evitar falhas no sistema, mas tais configurações devem ser aplicadas estrategicamente de acordo com a prioridade de cada lambda.

Vale também ressaltar que um serviço baseado em uma função que processa transações de maneira efêmera apresenta um atrito em termos de extração de métricas. A função só entra em

funcionamento rapidamente quando processa uma requisição, portanto seu processo não está perpetuamente ativo e não pode expor uma interface para coleta de informações que auxiliam no monitoramento de suas operações. Em contraste, para serviços HTTP que operam como um servidor (*"server"* ao invés de *"serverless"*), por exemplo, é comum expor métricas através de um caminho de URL, por exemplo, no endereço `/metrics`, e configurar um sistema de monitoramento para extrair (*scrape*) métricas periodicamente. Então, como o processo da função só existe enquanto a transação é processada, as métricas têm que ser emitidas e, se o número de invocações sair do controle, os *datapoints* dessas métricas podem ser rejeitados pelo sistema de armazenamento de métricas se o volume atingir o número máximo de dados emitidos por segundo. Estas são algumas ressalvas que SREs devem manter em mente enquanto fornecem suporte para serviços do tipo FaaS.

CaaS (Container-as-a-Service)

Semelhante ao PaaS, uma determinada versão da aplicação é distribuída em um *cluster*, porém, desta vez, em forma de réplicas de múltiplos contêineres criados a partir de imagens cujos *deployments* são orquestrados pela plataforma. Esta, por sua vez, também é encarregada de gerenciar o estado desta e de outras aplicações em uma frota de máquinas virtuais, que são os nós do *cluster* (*cluster nodes*). AWS EKS (Elastic Kubernetes Service), AWS ECS (Elastic Container Service), GCP GKE (Google Kubernetes Engine) ou Azure AKS (Azure Kubernetes Service) são alguns exemplos de plataformas CaaS. Todas as soluções de CaaS citadas aqui utilizam Kubernetes (também conhecido como "k8s") por trás das cortinas, com exceção do AWS ECS, portanto vamos focar em aspectos da arquitetura do Kubernetes e destacar algumas de suas funcionalidades que são mais relevantes para SREs.

As plataformas CaaS trabalham com tecnologias de conteinerização das quais a opção mais popular, o Docker, continua sendo

amplamente utilizada para fins de desenvolvimento e *deployment* de aplicações. As imagens construídas com Docker estão em conformidade com as especificações do OCI (Open Container Initiative), portanto são suportadas pela tecnologia de gerenciamento do ciclo de vida de contêineres "containerd" (lê-se: "container-D"), que é o padrão adotado pela indústria.

Imagens de contêineres são ótimas soluções para *builds herméticas* (como discutido anteriormente no capítulo 3), pois cada versão é gerenciada através da marcação (*tag*) de uma imagem, cada imagem testada em ambientes de Dev, QA e Pré-produção será idêntica à que será executada em Produção, o que traz mais confiabilidade e facilita o trabalho de SREs. A velocidade de *deployment* também é bastante satisfatória pois as réplicas de contêineres de uma mesma imagem são distribuídas em uma frota de nós do *cluster*, que são VMs pré-provisionadas que fazem parte do *cluster* e estão disponíveis para a execução imediata de contêineres.

Para explorarmos mais os conceitos de Kubernetes, temos que entender a sua abstração mais atômica: O *pod*. Um *pod* de Kubernetes é um agrupamento de contêineres que representa uma instância, ou réplica, de uma aplicação (no AWS ECS, entretanto, utiliza-se outro nome: *Task*). O *cluster* de Kubernetes inicia instâncias de tais agrupamentos de contêineres através de *deployments* (aplicações que não retém estado, ou *stateless*).

Existem outros tipos de recursos em Kubernetes que criam *pods*, como StatefulSets, Jobs ou CronJobs, mas, para simplificar a nossa jornada aqui, vamos focar apenas em *deployments*, já que representam a estratégia mais comum para executar uma aplicação em *clusters* de Kubernetes.

O *deployment* em Kubernetes é um componente lógico onde se define como a aplicação será distribuída e configurada uma vez que for instalada em um *cluster*. Por exemplo, operadores(as) podem definir o número de réplicas de Pods e um modelo (template) que

dita como cada réplica deve se comportar. *Deployments* são alocados a um determinado *namespace* (espaços lógicos dentro de um *cluster* para separar *deployments* de acordo com seu contexto). O template de um *deployment* contém a configuração de seus múltiplos contêineres, suas respectivas imagens, comandos, portas, volumes de disco, mapas de configurações, variáveis de ambientes etc., e cada recurso em um *cluster* de Kubernetes é criado a partir de um "manifesto", que é geralmente fornecido ao plano de controle do *cluster* (*control plane*) através de um arquivo YAML. Aqui está um exemplo de manifesto que é usado para criar um *deployment*:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: servico-x
  namespace: namespace-A
spec:
  replicas: 3
  selector:
    matchLabels:
      app: servico-x-deployment
  template:
    metadata:
      labels:
        app: servico-x-deployment
    spec:
      containers:
      - name: servico-x
        image: repositorio_docker/servico-x-imagem:v0.1
        ports:
        - containerPort: 8000
      - name: sidecar-metrics-exporter
        image: repositorio_docker/sidecar-metrics-exporter:v2.0
        ports:
        - containerPort: 9191
        volumeMounts:
        - name: servico-x-config
          mountPath: "/opt/servico-x/config"
```

```
    volumes:
    - name: servico-x-config
      configMap:
        name: servico-x-conf
```

Este exemplo ilustra um padrão comum em Kubernetes, que é a utilização de contêineres *sidecar*. Note que o manifesto declara dois contêineres na especificação de seu modelo. O *sidecar* é um outro contêiner que vive dentro do mesmo *pod* da aplicação, realizando tarefas auxiliares para expandir as capacidades operacionais da aplicação além da funcionalidade do contêiner principal. Neste exemplo, o nome deste contêiner *sidecar* fictício sugere que o *pod* possui um contêiner auxiliar que exporta métricas da aplicação.

A alocação de *pods* em um determinado nó do *cluster* se dá através do processo de agendamento (*scheduling*), que significa assinalar *pods* a um determinado nó do *cluster*. A plataforma gerencia nós do *cluster* (VMs), que podem ser agrupados com suas próprias configurações de escalonamento automático, e determina quais nós do *cluster* estão elegíveis para receber *pods* que estão na fila de agendamento de acordo com restrições e disponibilidade de recursos. Então múltiplos *pods* de diferentes aplicações podem compartilhar uma única máquina virtual, utilizando os recursos da infraestrutura de maneira mais eficiente e automaticamente "espalhando" as réplicas de *pods* de uma determinada aplicação em múltiplos nós do *cluster* e até mesmo provisionando novos nós para acomodar mais *pods* se a carga de trabalho computacional do *cluster* aumentar.

O operador pode aplicar configurações especiais baseadas em rótulos (*labels*) para distribuir réplicas de *pods* em nós específicos ou restrições de topologia para garantir que diferentes réplicas sejam alocadas para máquinas virtuais em zonas (*availability zones*) diferentes de uma mesma região (em outras palavras, em *data centers* diferentes), que é uma boa prática para alta disponibilidade (*high availability*) e confiabilidade de sites (mais informações em

https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/).

A imagem a seguir ilustra a arquitetura de um *cluster* de Kubernetes e como as réplicas de *pods* são distribuídas através de diferentes máquinas virtuais (nós do *cluster*):

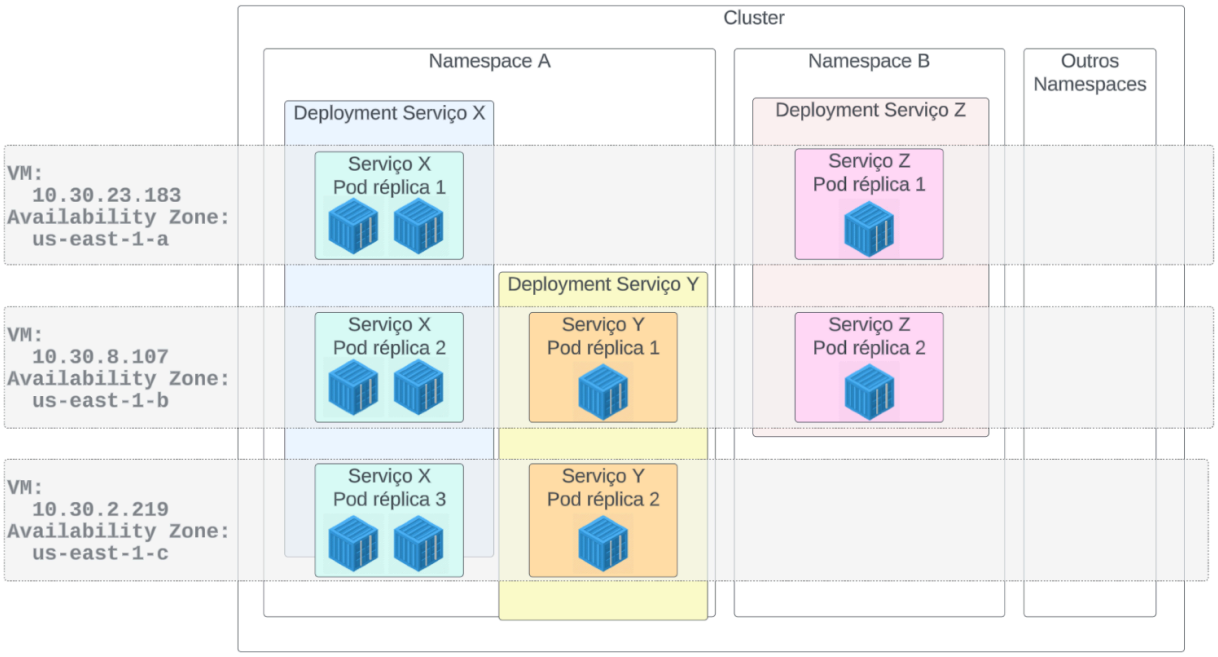

Figura 5.1: Arquitetura de um cluster Kubernetes

Então, neste modelo CaaS, SRE se beneficia de duas camadas de restauração automática, pois a plataforma mantém um contrato de escalabilidade similar ao PaaS no nível de nós do *cluster* e de réplicas de *pods*. Se um *pod* falhar, o *cluster* automaticamente vai agendar uma nova réplica para manter o contrato que define o número mínimo de instâncias de um determinado *deployment*. Se uma VM falhar e outras VMs estiverem disponíveis para acomodar novos *pods*, as réplicas de *pods* que estavam na VM indisponível vão migrar para outra VM saudável imediatamente, ou, se o *cluster* estiver exaurido de recursos, uma nova VM será provisionada para acomodar os *pods* e manter o número de réplicas de acordo com o contrato do *deployment*.

Para se aprofundar mais nas funcionalidades relacionadas à confiabilidade de uma plataforma CaaS, SREs devem se familiarizar com os conceitos de sonda de atividade (*liveness probe*) e sonda de prontidão (*readiness probe*). A sonda de atividades funciona da mesma maneira que os *health checks* executados periodicamente pelas plataformas PaaS: o operador pode configurar uma verificação contínua para informar ao plano de controle do *cluster* (*control plane*) quais das réplicas da aplicação estão operando com sucesso. Em aplicativos com um servidor HTTP, tais configurações são geralmente baseadas em uma requisição que é enviada a um endereço específico, por exemplo, uma rota para a url `/health` que executa uma ou mais operações que verificam o estado da aplicação e suas dependências. Se a sonda de atividades receber alguma resposta inválida de alguma réplica, esta instância da aplicação será substituída por um novo *pod* que será agendado para tomar o seu lugar. A sonda de prontidão, por outro lado, determina quando a aplicação está pronta para receber requisições ou processar transações, então esta sonda não é executada continuamente, apenas durante o estágio de inicialização (*bootstrapping*) de um *pod*.

Algumas aplicações precisam executar certas tarefas antes de serem expostas aos clientes para começarem a processar requisições. Algumas operações comuns são: estabelecer um *pool* de conexões com o banco de dados, obter segredos (senhas, chaves de API, tokens de autenticação etc.) ou até mesmo hidratar alguma estrutura de dados em memória e criar um cache interno para acelerar transações. Se a aplicação seguir boas práticas e implementar uma interface para indicar que a aplicação está pronta (*ready*), a sonda de prontidão vai instrumentar tal interface para garantir que uma nova instância só substituirá a anterior e receberá requisições quando a nova réplica fornecer respostas válidas.

Em caso de falha de inicialização ou se alguma dependência não for compatível com a nova versão e a verificação de prontidão conseguir capturar esse comportamento errôneo, o *deployment* por rotação

(*rolling deployment*) da nova versão será interrompido pelo plano de controle do *cluster* e aquela instância específica da aplicação será marcada como inválida (*unhealthy*) e não receberá nenhum tráfego. O *pod* será mantido no plano de controle para que o operador possa obter *logs* e observar os eventos do *cluster*, para fins de investigação.

A adoção de plataformas CaaS traz uma grande vantagem para SREs em termos de Observabilidade, porque o *cluster* é uma outra camada de abstração que facilita o monitoramento de eventos de maneira padronizada. Um *cluster* de Kubernetes emite várias métricas, para citar algumas:

- `container_cpu_usage_seconds_total`
- `pod_cpu_usage_seconds_total`
- `node_cpu_usage_seconds_total`
- `container_memory_working_set_bytes`
- `pod_memory_working_set_bytes`
- `node_memory_working_set_bytes`
- `prober_probe_total`

Como podemos ver, Kubernetes fornece uma variedade de métricas para medir a utilização de recursos a nível de contêiner, *pod* e nó (*kube node*), esta granularidade é bastante útil para monitorar a saúde do *cluster*. A métrica `prober_probe_total` possui outras informações como: `container`, `namespace`, `pod`, `pod_uid`, `probe_type` e `result` para informar o operador sobre quaisquer resultados em relação a sondas de atividades e prontidão. A lista completa de métricas pode ser encontrada aqui: https://kubernetes.io/docs/reference/instrumentation/metrics.

Com o padrão de *sidecar containers*, o operador pode adicionar mais contêineres ao *pod* de qualquer aplicação para obter métricas e expô-las para extração, ou mesmo exportar ou emitir métricas direto para sistemas de monitoramento. Este modelo de CaaS e *sidecar containers* tem um grande potencial para padronização já que podem ser adicionados para interceptar requisições e agir como

proxies, que controlam o tráfego de requisições para o contêiner principal da aplicação, tanto o ingresso quanto o egresso de todas as requisições em um *pod*. Estas ferramentas de controle de comunicações entre serviços são conhecidas como *Service Mesh*. As tecnologias de Service Mesh permitem que os desenvolvedores abstraiam os mecanismos e tecnologias relacionadas a métricas e deixem que os *sidecar containers* sejam encarregados desta responsabilidade, facilitando a vida de SREs com uma telemetria padronizada e previsível, o que torna o processo de monitoramento e triagem mais intuitivo e eficiente.

5.3 Outras plataformas

Existem outros tipos de serviço que podem ficar sob a guarda de SREs que operam em outras plataformas, como uma pipeline de dados (como Amazon Simple Workflow Service ou Google Workflows), sistemas de execução de modelos de Machine Learning (como Amazon SageMaker ou Google AutoML) ou de processamento em lote (como AWS Batch ou Google Batch). Estas outras plataformas são mais utilizadas em times de Ciência de Dados, que possuem serviços mais focados em Análise de Dados ou Aprendizado de Máquina.

SREs devem estar aptos a estudar e conhecer os componentes lógicos de qualquer plataforma que precisam de suporte, verificar seus requisitos operacionais e aplicar as práticas de confiabilidade: obter telemetria para observar SLIs e SLOs e gerenciar incidentes de acordo com a natureza de cada serviço e suas respectivas plataformas.

Serviços voltados a Análise de Dados, por exemplo, utilizam plataformas que devem ser avaliadas de acordo com os requisitos operacionais que aplicam mais escrutínio aos próprios dados. Como descrito no livro *Implementing Service Level Objectives* (Hidalgo,

2020), precisam de um foco adicional em requisitos de Confiabilidade de Dados (*Data Reliability*) como: Atualização, Completude, Consistência, Precisão, Validade, Integridade e Durabilidade. Não vamos nos aprofundar nestas outras plataformas de *runtime* já que os exemplos discutidos anteriormente já fornecem uma base de conhecimento e ressalvas que podem ser usadas como referência.

SREs devem analisar cada caso e, quando encontrarem diferentes plataformas que apresentem novos requisitos operacionais, colaborar com os engenheiros especializados nas tecnologias de tais plataformas para entender como alinhar suas funcionalidades e a telemetria disponível para observar indicadores. Com essa disciplina, SREs podem fornecer monitoramento, alarmes e relatórios que ajudam as entidades envolvidas a confirmar que as expectativas do negócio da organização sejam cumpridas (esses indicadores serão discutidos em detalhe no capítulo 9).

Agora que estamos familiarizados com alguns conceitos importantes relacionados a arquiteturas confiáveis, seus componentes de infraestrutura e exploramos algumas plataformas de *runtime* que são comumente utilizadas na indústria, no próximo capítulo vamos aprender como monitorar sistemas. Vamos falar sobre métricas, eventos, rastros, alarmes e *logs*. Vamos conhecer as tecnologias mais utilizadas para observar sistemas em produção e aprender porque Observabilidade é um conhecimento vital para SREs.

Parte 3: Observabilidade

Capítulo 6
Métricas

Métricas são dados emitidos para observar o comportamento de um serviço. Se considerarmos, por exemplo, um paciente em um hospital, podemos correlacionar a telemetria observada por médicos para fins de monitoramento como batimentos cardíacos, oxigênio ou temperatura, com as métricas emitidas por um servidor ou uma aplicação: CPU, Memória, Atividade de I/O (Input/Output) no Disco, Rx/Tx (Recebimento/Transmissão) via Rede, RPS (Requisições por Segundo), tempo de resposta ou erros. Estes são exemplos de métricas encontradas em um sistema de monitoramento.

SREs empregam métricas para observar o estado do sistema e para avaliar o impacto de quaisquer problemas com os serviços. Quando SREs respondem a um alarme, utilizam o monitoramento para fins investigativos, com o objetivo de encontrar relações de causa e efeito enquanto observam painéis que apresentam dados em forma de gráficos. Se um serviço está apresentando erros, a análise das métricas emitidas pela aplicação e sua infraestrutura pode indicar a causa-raiz do problema e nortear o próximo passo a ser tomado para aplicar uma mitigação.

Que tal um exemplo prático? Vamos supor que o seguinte alarme foi ativado às 3 da manhã: *Serviço X está produzindo muitos erros.* O(A) SRE *on-call* responde ao chamado, abre o seu laptop e visita o *dashboard* para iniciar a investigação. Ao observar os painéis relacionados ao tráfego e requisições HTTP, encontra o seguinte gráfico indicando que o número de respostas HTTP com o código 5xx teve um pico às 3 da manhã (os códigos de resposta HTTP 500 estão associados a erros que ocorrem do lado do servidor):

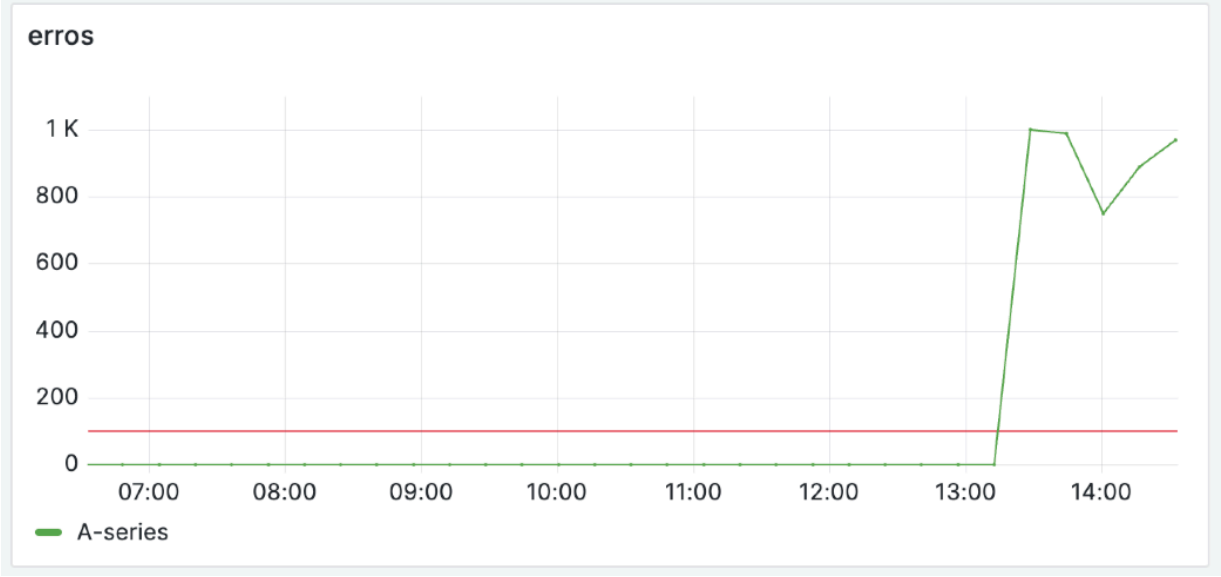

Figura 6.1: Gráfico de erros do serviço X

Logo depois, ao observar outros painéis associados às dependências utilizadas pelo serviço X que se encontram no mesmo *dashboard*, encontra um outro painel que apresenta o gráfico do percentual de espaço em disco livre do servidor de banco de dados. Este gráfico mostra que o disco deste servidor foi totalmente utilizado e que o espaço em disco foi exaurido por volta do mesmo horário em que os erros da aplicação começaram a ocorrer:

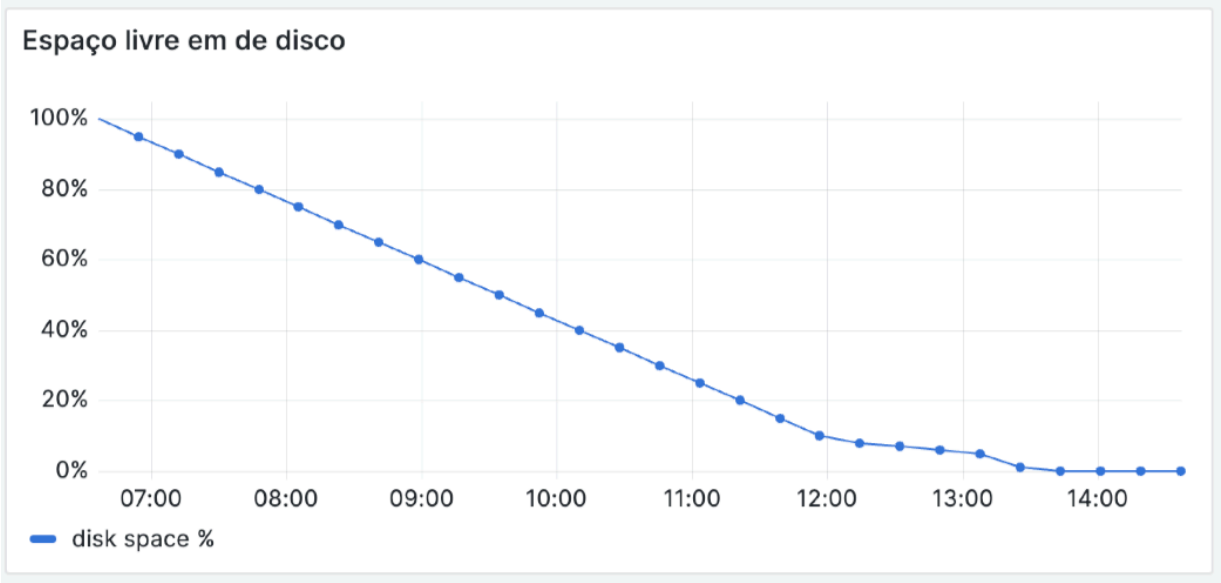

Figura 6.2: Percentual de espaço em disco livre chegando a zero

Então, conclui-se que a causa-raiz deste alarme foi a falta de espaço em disco no servidor de banco de dados. Uma mitigação simples para este caso é redimensionar o disco para obter mais espaço e restaurar o serviço X para um estado em que possa continuar a servir requisições e inserir informações no banco de dados. Mas como o espaço será consumido novamente de acordo com o padrão observado, SRE *on-call* pode redimensionar o disco apenas para restaurar o serviço, mesmo que não seja uma solução permanente, e manter o sistema saudável o suficiente para que possa voltar a dormir e esperar até a manhã seguinte quando os responsáveis pelo serviço X possam iniciar uma investigação mais criteriosa.

6.1 Os 4 sinais de ouro de SRE

Existe uma infinidade de métricas que podem ser utilizadas para monitoramento, porém SREs focam em 4 sinais que expressam como os usuários percebem a qualidade de um serviço:

- **Latência:** a lentidão experimentada pelo usuário. Este é o sinal utilizado para medir a performance de um serviço em termos de velocidade. Serviços de baixa latência produzem respostas em um tempo razoável.
- **Tráfego:** o volume de transações que o serviço está acomodando em um determinado momento. O número de requisições a um website, por exemplo.
- **Erros:** falhas no serviço. Causadas por bugs no código, dependências defeituosas, ou interrupções causadas por algum problema na infraestrutura do sistema.
- **Saturação:** exaustão de algum recurso. Tipicamente relacionada com cenários onde uma aplicação atinge algum limite de utilização de algum recurso, como a capacidade de transferência de dados via rede, o consumo de memória ou CPU de um servidor, ou quando todo o espaço de um disco é ocupado, como descrito anteriormente.

Estes sinais servem como uma orientação para que SREs escolham as métricas certas para incluir em suas estratégias de Observabilidade, monitorando serviços a partir de sua telemetria mais crítica, sob o ponto de vista dos clientes, e criando um número menor de alarmes (vamos falar mais sobre essas estratégia no capítulo 9).

6.2 Bancos de dados de série temporal

Para nos aprofundarmos mais nas tecnologias de emissão, armazenamento e renderização de dados obtidos através de métricas, vamos explorar duas tecnologias de bancos de dados de série temporal (*Time Series Databases*): Graphite (https://graphiteapp.org) e Prometheus (https://grafana.com/oss/prometheus).

Para conduzir os experimentos descritos nas próximas seções, nós vamos precisar da ferramenta de gerenciamento e execução de contêineres chamada *Docker*. Instruções para sua instalação podem ser encontradas em https://docs.docker.com/engine/install. Os comandos executados neste livro utilizam programas que se encontram no terminal do sistema operacional Linux, portanto usuários Windows devem utilizar ferramentas para emular estes programas, como o *Windows Subsystem for Linux* (WSL), por exemplo.

Graphite

Graphite é uma tecnologia de código aberto para armazenamento de métricas e geração de gráficos. As métricas podem ser enviadas para o Graphite em texto plano (*plain text*) no seguinte formato:

`<métrica> <valor numérico> <data/hora>.`

Para brincar um pouco com Graphite, inicie um contêiner Docker localmente:

```
docker run --rm -d --name graphite -p 2003-2004:2003-2004
graphiteapp/graphite-statsd
```

Para saber mais sobre o comando `docker run`, você pode encontrar a documentação completa em https://docs.docker.com/engine/reference/run/.

Uma vez que o contêiner `graphite` estiver em execução, envie algumas métricas para testar suas funcionalidades.

```
echo "meu.contador.exemplo.infoA 1.0 $(date +%s)" | nc -vv localhost 2003
```

A instrução acima está executando o comando `echo` (ferramenta de interface de linha de comando) encontrado em sistemas UNIX/LINUX para imprimir mensagens, e a barra vertical `|`, conhecida como *pipe*, está redirecionando a saída do comando `echo` para outro programa chamado *netcat* (`nc`) que é um utilitário para transmitir e

receber dados através de conexões de rede. Para leitores(as) que estão utilizando o sistema operacional Windows, recomenda-se a instalação de algum programa que permita a execução de tais ferramentas de linhas de comando como o *Cygwin* ou o *Windows Subsystem for Linux* (WSL).

Vamos estudar a estrutura lógica da informação que está sendo enviada. No Graphite, uma métrica é composta pelos seguintes elementos:

- Nome ou *série*: "meu.contador.exemplo.infoA"
- Valor da métrica: 1.0
- Data/Hora: O comando `$(date +%s)` produz um número semelhante a `1704067200`, que representa um formato de data/hora universal baseado no número de segundos desde 1970-01-01 00:00:00 UTC (sigla em inglês para *Universal Time Coordinated*, ou Tempo Universal Coordenado em português).

O comando deve imprimir a seguinte resposta em seu terminal:

```
Connection to localhost port 2003 [tcp/*] succeeded!
```

Essa é a resposta do utilitário *netcat* informando o usuário que a conexão com o destino `localhost` na porta `2003` foi efetuada com êxito. Em outras palavras, a comunicação com o contêiner que foi criado a partir da imagem `graphiteapp/graphite-statsd` foi estabelecida.

O Graphite possui funções que operadores(as) podem utilizar para agregar os valores de suas séries. Vamos submeter mais métricas para o banco de dados de séries temporais para ilustrar a funcionalidade de agregação:

```
echo "minha.metrica.amostra.a 5.0 $(date +%s)" | nc -vv localhost 2003
echo "minha.metrica.amostra.b 10.0 $(date +%s)" | nc -vv localhost 2003
echo "minha.metrica.amostra.c 6.0 $(date +%s)" | nc -vv localhost 2003
```

Espere alguns segundos e repita esse procedimento algumas vezes. Para um melhor resultado o banco precisa de uma amostragem

mínima de dados para realizar cálculos apropriadamente. Depois de executar esses comandos aguardando alguns segundos entre as próximas rodadas de envio de métricas, vamos ver uma agregação de dados em ação ao executar a função `avg` (*Average* ou Média, em português) e observar os resultados.

O Graphite possui uma API de renderização (`/render`) para retornar os dados de séries temporais correspondente às métricas definidas em uma consulta (*query*). Daqui em diante vamos referenciar o resultado de tais consultas apenas como *datapoints*. O comando a seguir mostra os resultados de uma agregação de múltiplas séries (amostras `a` , `b` e `c`) utilizando a API de renderização (*Render API*):

```
curl -s -H "Content-Type: application/json" -X GET
"http://localhost/render?
target=avg(minha.metrica.amostra.*)&from=-10min&format=json"
```

Vamos analisar o alvo (*target*) deste exemplo de uma consulta ao banco: `avg(minha.metrica.amostra.*)` . Repare que o último fragmento da série não é especificado e vemos apenas um asterisco (`*`), o que significa que não estamos especificando qual amostra está sendo selecionada, apenas utilizamos a notação `*` para expressar que a consulta deve incluir todas as variações de fragmentos da série que se encontram naquele ponto, ou seja:

- `minha.metrica.amostra.a`
- `minha.metrica.amostra.b`
- `minha.metrica.amostra.c`

Desta forma, todos os valores correspondentes a essas séries devem ser agregados no cálculo da Média.

Sobre os outros parâmetros encontrados na URL: o parâmetro `from=-10min` está especificando que queremos a agregação de todos os *datapoints* dos últimos 10 minutos (você pode alterar esse número e observar diferentes resultados). O último parâmetro da URL é o `&format=json` , que define o formato da resposta produzida

pela API que está em JSON (vários outros formatos são suportados: CSV, PNG, PDF etc.).

Para formatar a resposta JSON, você pode instalar a ferramenta **jq** (https://jqlang.github.io/jq/download/), que é um pequeno utilitário para processar documentos JSON. Com o `jq` instalado, você pode executar comandos que retornam dados em formato JSON e ver o resultado de uma maneira mais legível selecionando blocos e parâmetros específicos. No exemplo a seguir, a instrução `jq` está acessando o primeiro elemento da lista de resultados da consulta `.[0]`, iterando sob os `datapoints[]` e filtrando apenas os valores que não são nulos (`!= null`):

```
curl -s -H "Content-Type: application/json" -X GET
"http://localhost/render?
target=avg(minha.metrica.amostra.*)&from=-10min&format=json" |  jq '.
[0].datapoints[] | select(.[0] != null)'

[
  7,
  1694401030
]
```

Observe que os *datapoints* são estruturados em forma de *tuplas* (conjunto de informações, geralmente em pares) contendo o valor de um dado de uma métrica com sua respectiva Data/Hora (*timestamp*). Na resposta do comando, o primeiro elemento dessa lista é o valor do resultado da agregação `7`, enquanto o segundo representa a Data/Hora correspondente ao mesmo `1694401030`. Essa distribuição de dados através de uma linha do tempo é ideal para criar gráficos em painéis para fins de monitoramento e correlação com outros eventos. Esta é a diferença fundamental entre bancos de dados relacionais e bancos de dados de séries temporais, as informações registradas em linhas, colunas e tabelas de bancos de dados convencionais não são ideais para a renderização de agregações estatísticas distribuídas cronologicamente.

As séries são basicamente conjuntos de fragmentos separados por um ponto (.) que são utilizadas para armazenar *datapoints* em uma linha do tempo. Quando métricas são emitidas, permitem que operadores(as) possam definir diferentes pedaços de informação de acordo com os fragmentos de cada série e seus respectivos valores. Por exemplo, imagine uma métrica com o seguinte nome: `xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.200` , no qual se define uma sequência de informações como `<nome da organização>.<nome do serviço>.<datacenter>.<ambiente>.<outros fragmentos da métrica>` . Vamos ver este conceito em ação e executar os comandos a seguir para emitir métricas representando um simples contador (*counter*), simulando um serviço que está processando requisições HTTP cujo número de respostas deve ser observado por um sistema de monitoramento:

```
echo "xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.200 1.0
$(date +%s)" | nc -vv localhost 2003
echo "xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.300 1.0
$(date +%s)" | nc -vv localhost 2003
echo "xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.400 1.0
$(date +%s)" | nc -vv localhost 2003
echo "xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.500 1.0
$(date +%s)" | nc -vv localhost 2003
```

Agora vamos realizar uma consulta com `http.respostas.*` para conferirmos o resultado:

```
curl -s -H "Content-Type: application/json" -X GET
"http://localhost/render?target=xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.*&from=-3min&format=json" | jq '.'
[
  {
    "target": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.200",
    "tags": {
      "name": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.200"
    },
```

```
    "datapoints": [
      [
        1,
        1710029200
      ]
    ]
  },
  {
    "target": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.300",
    "tags": {
      "name": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.300"
    },
    "datapoints": [
      [
        1,
        1710029220
      ]
    ]
  },
  {
    "target": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.400",
    "tags": {
      "name": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.400"
    },
    "datapoints": [
      [
        1,
        1710029220
      ]
    ]
  },
  {
    "target": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.500",
    "tags": {
```

```
      "name": "xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.500"
    },
    "datapoints": [
      [
        1,
        1710029220
      ]
    ]
  }
]
```

Repare que a consulta retorna múltiplas séries com seu respectivos *datapoints*, então, se quisermos contar o número total de respostas HTTP agregando todos os *datapoints* de todas as séries encontradas no fragmento que contém `*`, precisamos da função de agregação `sumSeries`:

```
% curl -s -H "Content-Type: application/json" -X GET
"http://localhost/render?target=sumSeries(xyz.loja-api.datacenter-
a2.producao.trafego.http.respostas.*)&from=-10min&format=json" | jq '.
[0].datapoints[] | select(.[0] != null)'
[
  4,
  1710029890
]
```

Isso é bastante útil para observar um sistema já que séries com fragmentos bem-definidos facilitam o monitoramento de diferentes segmentos da infraestrutura e funcionalidades de uma aplicação.

Outra prática comum é a criação de métricas cujos valores podem ser utilizados em conjunto para cálculos matemáticos. Podemos ilustrar isso criando uma métrica que expressa o valor total da memória disponível em um cluster `xyz.loja-api.datacenter-a2.producao.cluster.memoria.total` e uma outra métrica que contém um valor que deve oscilar de acordo com o consumo de memória das instâncias de uma aplicação `xyz.loja-api.datacenter-`

`a2.producao.replica-1.memoria.consumida`. Aqui estão os comandos para alimentar o Graphite com esses *datapoints*:

```
echo "xyz.loja-api.datacenter-a2.producao.cluster.memoria.total
64000000000.0 $(date +%s)" | nc -vv localhost 2003
echo "xyz.loja-api.datacenter-a2.producao.replica-1.memoria.consumida
16000000000.0 $(date +%s)" | nc -vv localhost 2003
```

E aqui está o comando para verificar o percentual de uso de memória, correlacionando essas duas métricas:

```
curl -s -H "Content-Type: application/json" -X GET
"http://localhost/render?target=scale(divideSeries(xyz.loja-
api.datacenter-a2.producao.replica-1.memoria.consumida,xyz.loja-
api.datacenter-
a2.producao.cluster.memoria.total),100)&from=-3min&format=json" | jq '.
[0].datapoints[] | select(.[0] != null)'
[
  25,
  1710025130
]
```

Note que o comando contém uma função que aplica uma operação de divisão entre os valores de `memoria.consumida` e `memoria.total` e depois aplica uma função de `scale` que multiplica o resultado por `100`, basicamente replicando a seguinte equação: `(memoria.consumida / memoria.total) * 100`. Essa consulta produz o resultado `25`, o que significa que, neste cenário em que simulamos métricas emitidas por uma aplicação fictícia, ela estaria consumindo `16000000000` dos `64000000000` bytes disponíveis de memória, o que corresponde a 25%.

Esse valor pode oscilar à medida que novos *datapoints* são emitidos para o Graphite. Faça um pequeno experimento e continue enviando ambas as métricas alterando o valor da métrica `xyz.loja-api.datacenter-a2.producao.replica-1.memoria.consumida` e execute a consulta novamente para ver mudanças no percentual de consumo de memória.

Este é apenas um exemplo de como métricas e funções de agregações de bancos de dados de séries temporais podem nos ajudar a obter informações úteis sobre o estado de um sistema.

Para conhecer mais funções oferecidas pelo Graphite, veja sua documentação: https://graphite.readthedocs.io/en/latest/functions.html#module-graphite.render.functions.

Por enquanto, estamos explorando essas tecnologias apenas com ferramentas de linha de comando, mas, no dia a dia, SREs utilizam interfaces web em navegadores para observar gráficos gerados a partir de *datapoints* distribuídos na linha do tempo de vários painéis, selecionando a Data/Hora e ajustando a janela de tempo de acordo com os eventos que devem ser investigados. Vamos falar mais sobre *dashboards* e painéis de visualização de métricas nas próximas seções deste capítulo.

Agora vamos explorar outra opção de banco de dados de séries temporais: Prometheus.

Prometheus

Prometheus também é uma tecnologia de código aberto que trabalha com coleta, armazenamento e renderização de métricas. Porém, ele opera de uma maneira diferente do Graphite pois obtém métricas através de extração (*scraping*) em vez de receber métricas de clientes. Enquanto os serviços enviam métricas para o Graphite, quando se utiliza Prometheus, os serviços expõem um endereço em sua API HTTP (por exemplo: `/metrics`) contendo métricas no formato esperado, que são absorvidas pelo próprio sistema de extração de métricas do Prometheus em uma frequência configurável. Então, diz-se que existem sistemas de monitoramento passivos, que apenas recebem métricas sem se preocupar com a origem delas (como Graphite), e existem os ativos, que obtêm métricas extraindo dados de aplicações (Prometheus).

Para ver o Prometheus em ação, vamos primeiro iniciar o seguinte contêiner Docker:

```
docker run --rm -d --name=grafana -p 3000:3000 grafana/grafana
```

Este comando vai iniciar uma instância da ferramenta de monitoramento **Grafana** (https://grafana.com/). Quando o contêiner `grafana` é inicializado, ele já expõe métricas em formato Prometheus no endereço `/metrics` e você pode observar essas métricas em seu browser ao navegar para: http://localhost:3000/metrics.

Aqui está um exemplo de métrica:

```
curl -s "http://localhost:3000/metrics" | grep
"grafana_api_response_status_total"

# HELP grafana_api_response_status_total api http response status
# TYPE grafana_api_response_status_total counter
grafana_api_response_status_total{code="200"} 2
grafana_api_response_status_total{code="404"} 0
grafana_api_response_status_total{code="500"} 0
grafana_api_response_status_total{code="unknown"} 1
```

Veja que as métricas em formato Prometheus são bem diferentes comparadas com as métricas do Graphite e, em vez de serem enviadas ao banco de dados de séries temporais, elas são expostas e têm seus valores modificados à medida que a aplicação realiza alguma operação.

É hora de um novo experimento: visite a URL http://localhost:3000 para acessar a aplicação Grafana, utilizando `admin` e `admin` como usuário e senha, e atualize a página em seu browser algumas vezes. Agora execute o comando anterior para obter a lista de métricas novamente e, desta vez, repare que o valor da métrica `grafana_api_response_status_total{code="200"}` será incrementado de acordo com o número de vezes em a aplicação processou uma requisição HTTP.

Uma métrica do Prometheus tem a seguinte estrutura:

- Duas linhas de texto iniciadas por uma cerquilha `#`
  - `# HELP` : contém um texto que ajuda a descrever o propósito da métrica
  - `# TYPE` : define o tipo da métrica (existem quatro tipos: *Counter*, *Gauge*, *Histogram* e *Summary*)
- O nome da métrica em minúsculas com fragmentos de texto separados por um traço-baixo (*underscore*) seguido de um conjunto de rótulos ou, do inglês: *labels*, cercados por chaves `{}` . Vamos utilizar o termo em inglês *labels* para nos referir aos rótulos de agora em diante.
  - Cada *label* possui uma chave (*key*) e um valor (*value*), como podemos ver no exemplo: `grafana_api_response_status_total{code="200"}` , onde encontramos a chave `code` e o valor `200` , que se refere ao código de resposta HTTP.

Agora que nos familiarizamos com as métricas do Prometheus, vamos ver como estas métricas podem ser extraídas. Porém, antes de prosseguirmos com o próximo experimento, precisamos criar uma rede Docker para permitir a comunicação entre os contêineres `grafana` e `prometheus` .

Execute o comando a seguir para interromper a execução do contêiner `grafana` :

```
docker stop grafana
```

E com este outro comando, vamos criar uma rede Docker para estabelecer uma comunicação entre os contêineres:

```
docker network create --driver bridge sre-test-network
```

Podemos então iniciar o contêiner `grafana` novamente mas, desta vez, vamos definir uma configuração de rede para o contêiner com o

parâmetro `--network` :

```
docker run --rm -d --name=grafana -p 3000:3000 --network sre-test-network
grafana/grafana
```

Para configurar a extração de métricas e vê-las em ação, temos que criar um arquivo de configuração e iniciar um contêiner com o Prometheus para extrair (*scrape*) as métricas em uma certa cadência configurável e executar um servidor HTTP com um console de interface web para consultas.

Primeiro, vamos criar um arquivo chamado `prometheus.yml` contendo o seguinte código:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: "extraindo-metricas-do-grafana"
    static_configs:
      - targets: ["grafana:3000"]
```

A primeira parte da configuração, `global` , refere-se a parâmetros que se aplicam a todas as configurações de extração definidas no resto do documento, o que significa que as métricas serão extraídas a cada 15 segundos (de acordo com o `scrape_interval`). Logo abaixo temos uma lista de configurações de extração (`scrape_configs`), que contém apenas uma instância que possui um nome (`job_name`) e uma lista de configurações estáticas (`static_configs`). Este exemplo define uma lista de alvos (`targets`), utilizada para definir URLs arbitrárias, e possui apenas um item: `grafana:3000` , que corresponde ao nome do outro contêiner e à porta da aplicação Grafana dentro da rede Docker. Note que não é necessário definir o caminho de URL `/metrics` já que este é o valor padrão para esta configuração.

Com esta configuração definida, vamos finalmente iniciar o contêiner `prometheus` :

```
docker run --rm -d --name prometheus -p 9090:9090 --network sre-test-network -v $(PWD)/prometheus.yml:/etc/prometheus/prometheus.yml --entrypoint sh prom/prometheus -c "/bin/prometheus --config.file=/etc/prometheus/prometheus.yml"
```

Você pode verificar a lista de contêineres com o comando `docker ps -a`. Se todos os passos foram executados com sucesso, você deve ver os dois contêineres: `grafana` e `prometheus` em atividade (*Status: Up*).

Visite a URL do Grafana com o seu browser (http://localhost:3000) e atualize a página algumas vezes para incrementar a métrica que corresponde ao contador de respostas HTTP.

Agora visite a URL http://localhost:9090 para acessar o console web do Prometheus. Encontre o campo de texto onde se definem as expressões (*Expressions*), execute a seguinte consulta: `sum(grafana_api_response_status_total{code="200"})` (aperte "Enter" para submeter a consulta) e clique na aba *Graph* para exibir um gráfico com os *datapoints* desta métrica.

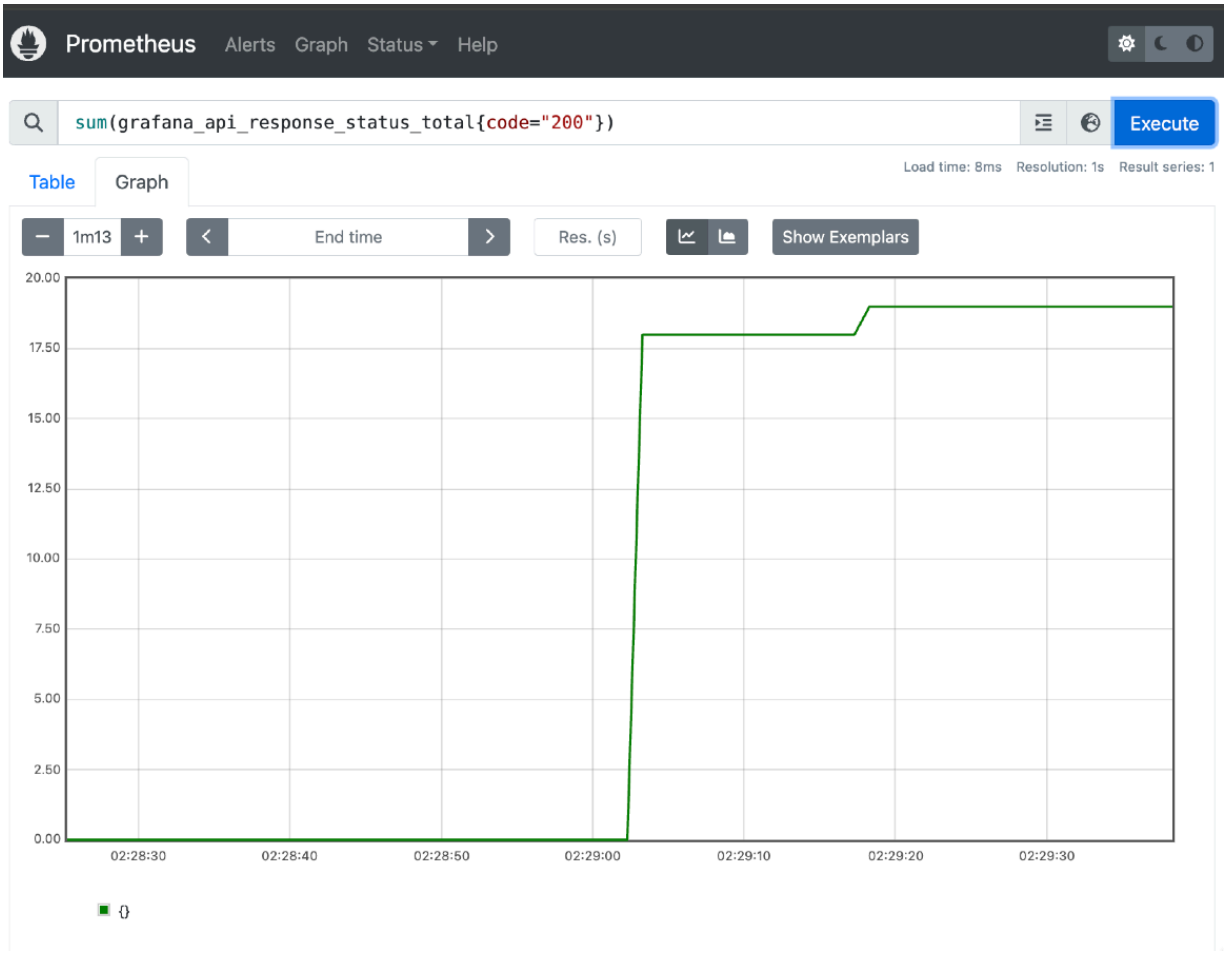

Figura 6.3: Um exemplo de gráfico produzido com datapoints de uma métrica Prometheus

Visite a URL http://localhost:3000 mais uma vez para acessar a aplicação Grafana e atualize a página algumas vezes (para gerar novas métricas que correspondem a requisições bem-sucedidas). Ao retornar para o console do Prometheus, execute a consulta mais uma vez e veja o gráfico exibindo novos *datapoints*. Tente selecionar uma janela de tempo para focar nos resultados mais recentes, por exemplo: os últimos 15 ou 5 minutos.

Mova o ponteiro do mouse sobre os *datapoints* no gráfico e correlacione as alterações nos valores de cada ponto com o horário associado aos mesmos. O eixo Y do gráfico mostra as faixas de valores (ex.: 0, 5, 10, 15, 20, ...) enquanto o eixo X apresenta os espaços de data e hora (também conhecidos como *timestamps*).

Visite a URL `http://localhost:3000/nao-existe` para tentar acessar uma página que não existe, repita esta operação algumas vezes e execute a seguinte consulta no console do Prometheus para comparar o número de respostas HTTP separadas por seus respectivos códigos: `sum by (code) (grafana_api_response_status_total)`.

Repare que o número encontrado nesses *datapoints* é incrementado para cada requisição, mas não mostra exatamente quantas requisições o serviço está processando no momento. Isso acontece porque a métrica `grafana_api_response_status_total` é do tipo *contador* (*counter*), portanto é incrementada monotonicamente, ou seja, o valor está sempre subindo até que seja resetado de volta para zero.

Quando monitoramos um serviço, estamos interessados em observar quantas transações estão sendo processadas em espaços de tempo específicos, como 1 minuto ou 30 segundos, para ajudar operadores(as) a medir o tráfego de uma determinada aplicação. Para monitorar as mudanças nos valores entre esses *datapoints*, utiliza-se a função `rate` da seguinte forma:
`sum(rate(grafana_api_response_status_total[30s]))`.

A função `rate` requer um espaço de tempo, que deve ser definido entre colchetes (`[]`) e cujos valores são expressado com unidades como `s` (segundos), `m` (minuto), `h` (hora). A faixa de tempo define a quantidade de *datapoints* utilizados no cálculo da taxa média de incremento, que deve estar alinhada com a frequência em que as métricas são extraídas. Por exemplo: se a configuração do Prometheus contém `scrape_interval: 15s`, ao processar uma consulta com `sum(rate(grafana_api_response_status_total[20s]))`, o gráfico não vai renderizar linhas contínuas pois espaços de 20s não compreendem 2 ciclos de extração e o cálculo da média precisa de, pelo menos, duas amostragens.

O Prometheus oferece outras funções em sua linguagem de consulta: *PromQL* (abreviação de *Prometheus Query Language*), que podem ser encontradas em sua documentação oficial: https://prometheus.io/docs/prometheus/latest/querying/basics/.

Agora que aprendemos um pouco mais sobre métricas do tipo *contador* (*counter*), vamos explorar os outros tipos de métricas do Prometheus: Medidor (conhecido como *gauge*), Histograma (*histogram*) e Sumário (*summary*).

O *gauge* é a métrica que, ao contrário do *contador*, possui valores que podem subir ou descer (como o velocímetro do painel de um carro).

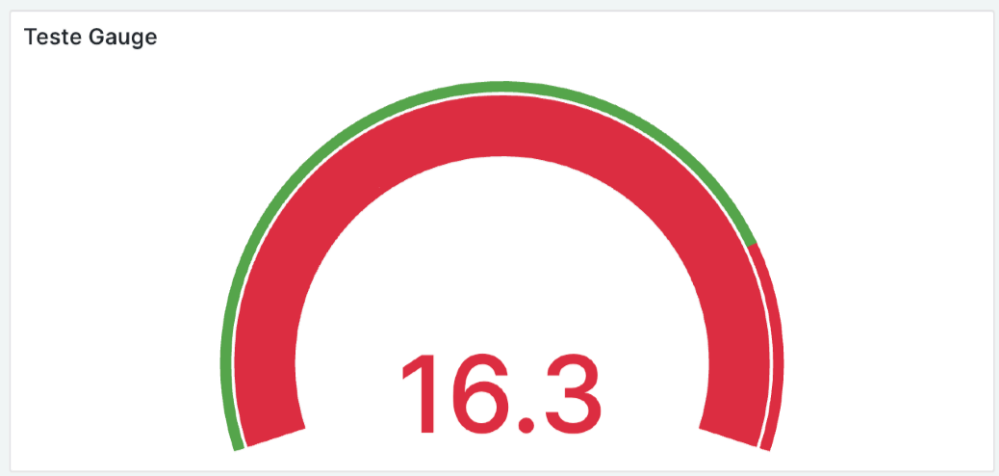

Figura 6.4: Ilustrando a métrica do tipo "Gauge"

Na aplicação Grafana, se explorarmos novamente a lista de métricas em http://localhost:3000/metrics, vamos encontrar várias métricas do tipo *gauge*, como:

```
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 147
```

Essa métrica produzida pelo Grafana ajuda os operadores(as) a monitorarem o número de rotinas de sua aplicação. Os *datapoints* de tais métricas podem ser agregados com funções como `avg` (média), `max` (máximo) ou `min` (mínimo) para métricas que possuem *labels*, agregando valores de um vetor em múltiplas dimensões, e para cálculos de valores distribuídos em um espaço de tempo utiliza-se as funções `avg_over_time`, `max_over_time` ou `min_over_time`. Exemplo: `max_over_time(go_goroutines[5m])`.

As métricas do tipo histograma (*histogram*) e sumário (*summary*) são bem semelhantes, pois ambas apresentam uma distribuição de medidas em que cada faixa é populada de acordo com os valores emitidos pela aplicação, o que facilita o monitoramento pois operadores(as) podem isolar agregações de faixas específicas para observar eventos anômalos como um pico na latência de um serviço.

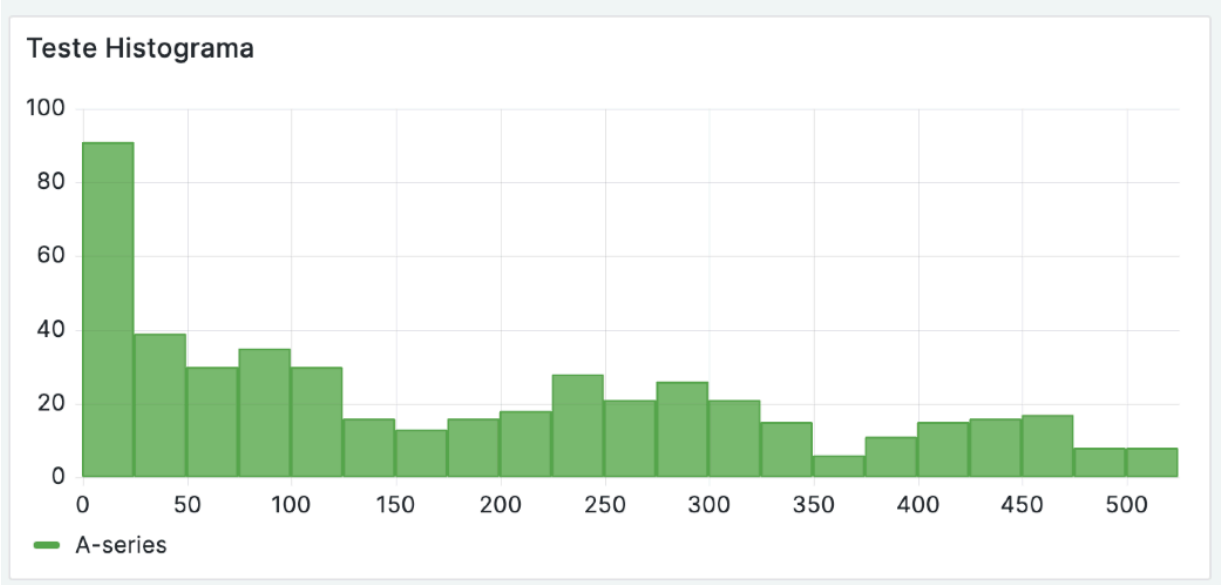

Figura 6.5: Ilustrando a métrica do tipo "Histograma"

A diferença básica entre o histograma e o sumário é que o histograma distribui os valores de acordo com os *datapoints* extraídos pela aplicação através de um *label* chamado `le` que significa "menor ou igual" (do inglês *less than or equal to*), e os agrupa de acordo com suas respectivas faixas, enquanto o sumário distribui os valores de acordo com *quantils* que são calculados na aplicação, onde as bibliotecas compatíveis com Prometheus preparam as métricas do lado cliente.

Quantils (*quantiles*) são designações de estatística, assim como o *percentil* (*percentile*), que dividem valores em conjuntos de dados ordenados. Os quantis são expressos em faixas de 0 a 1.0 e os percentis vão de 0 a 100. Se considerarmos uma lista de números como `[1, 4, 8, 3, 9, 2, 7, 6, 5]`, o *p50* e *q50* (o quinquagésimo percentil e quantil) é o valor `5` e o *p90* e *q90* (o nonagésimo percentil e quantil) é `8.2`. O percentil *p90* é calculado através da seguinte equação: `( percentil * (tamanho da amostra - 1) / 100 ) + 1`, ou seja: `( 90 * (9 - 1) / 100) + 1`.

Para ilustrar, aqui está um exemplo de um histograma utilizado para monitorar o tempo de resposta das requisições processadas pelo Grafana:

```
# HELP grafana_http_request_duration_seconds Histogram of latencies for
HTTP requests.
# TYPE grafana_http_request_duration_seconds histogram
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.005"} 1
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.01"} 3
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.025"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.05"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.1"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.25"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="0.5"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="1"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="2.5"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="5"} 5
```

```
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="10"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="25"} 5
grafana_http_request_duration_seconds_bucket{handler="public-
assets",method="GET",status_code="404",le="+Inf"} 5
grafana_http_request_duration_seconds_sum{handler="public-
assets",method="GET",status_code="404"} 0.045836667
grafana_http_request_duration_seconds_count{handler="public-
assets",method="GET",status_code="404"} 5
```

Essas métricas distribuem uma contagem do número de requisições cujas durações se encaixam em cada variação da *label* `le`, isso quer dizer que `3` de um total de `5` requisições, que tinham como alvo a url `public-assets` utilizando o verbo HTTP `GET` e que obtiveram o código de resposta `404`, tiveram a duração abaixo ou igual a `10ms` (de acordo com `le="0.01"`), enquanto apenas `1` de `5` requisições foi processada abaixo de `5ms` (`le="0.005"`).

Essa distribuição de valores é utilizada para "plotar" ou renderizar o gráfico de um histograma para ilustrar as amostras de acordo com os intervalos (também conhecidos como *bins* ou *buckets*). Tais intervalos são predefinidos de acordo com a biblioteca da linguagem de programação escolhida pelos operadores(as), porém podem ser configurados no código e devem estar alinhados com os objetivos de monitoramento de um serviço (caso seja necessário observar um limiar de duração mais específico).

Podemos usar um histograma para monitorar a latência da aplicação através de consultas *promQL* utilizando a seguinte função: `histogram_quantile` combinada com a taxa de mudança da duração de uma requisição em uma determinada faixa de tempo, por exemplo: `histogram_quantile(0.99, rate(http_request_duration_seconds_bucket{api="websiteX"}[5m]))`. Essa é uma prática comum para obter o *percentil* p99 do tempo de resposta de um serviço.

Para observarmos o *percentil* p99 da latência da aplicação Grafana ao processar requisições HTTP utilizando o verbo `GET`, vamos utilizar a seguinte consulta:

```
histogram_quantile(0.99,
rate(grafana_http_request_duration_seconds_bucket{method="GET"}[5m]))
```

Para visualizar a latência de cada combinação de caminho de URL e verbo HTTP, podemos utilizar esta outra consulta:

```
histogram_quantile(0.99, sum by (handler, method, le)
(rate(grafana_http_request_duration_seconds_bucket{method="GET"}[10m])))
```

Aqui está uma consulta interessante que pode ser utilizada para monitorar o percentual de requisições que obedece a um determinado limiar:

```
 sum(rate(grafana_http_request_duration_seconds_bucket{le="0.01"}[5m])) /
sum(rate(grafana_http_request_duration_seconds_count[5m]))
```

Essa consulta vai retornar algum valor entre `0.0` e `1.0` calculando a divisão entre o número de requisições cuja duração está abaixo de `10ms` e o total de requisições. Este percentual pode ser utilizado para informar quantos usuários estão impactados por alguma anomalia onde o serviço passa a operar de forma mais lenta.

Esses são apenas alguns exemplos de métricas e consultas com as quais SREs devem estar bem familiarizados. Porém, existe outra funcionalidade importante do Prometheus que SREs devem conhecer, que é particularmente útil para aplicações das quais não se pode extrair métricas periodicamente. Nos exemplos anteriores, exploramos métricas da aplicação Grafana que foram coletadas em ciclos de extração executados a cada 15 segundos de acordo com as configurações de extração (`scrape_config`), como visto anteriormente. Para programas onde a extração não é viável, como operações de lote (*Batch*) ou programas efêmeros como Lambdas, por exemplo, o Prometheus possui um componente chamado *Pushgateway* que pode ser instalado separadamente e permite que tais programas enviem métricas através de requisições HTTP. Esta é

uma alternativa que deve ajudar SREs a monitorarem outros tipos de serviços que não operam como serviços de API convencionais.

Nesta seção exploramos bancos de dados de séries temporais, os tipos de métricas e como elas podem ser emitidas ou coletadas utilizando diferentes tecnologias. No capítulo 9 vamos aprender mais sobre aplicações práticas de monitoramento com foco em objetivos de nível de serviço, que é um conceito essencial para a adoção de SRE.

6.3 Cuidados com métricas

Ao definir uma métrica, é recomendado não abusar do número de *labels* para evitar a carga computacional de lidar com métricas que possuem muitas dimensões, ou seja, muitos valores únicos para um grande número de *labels* com seus pares de chave e valor.

Para ilustrar, vamos supor que uma aplicação de uma loja virtual possui uma métrica `user_login_total` e um(a) desenvolvedor(a) decide criar um *label* para monitorar quantas vezes usuários estão interagindo com o serviço de autenticação do website. Vamos dizer que esta *label* possui uma chave `email` e os valores correspondem ao endereço de email de cada um dos usuários desta loja virtual. Esse é um cenário clássico de alta cardinalidade (*high cardinality*), pois o banco de dados terá que processar e armazenar uma série temporal (*time series*) para cada combinação única da métrica e os respectivos pares de chave e valor de cada *label*, exigindo um custo adicional de memória, CPU, disco e volume de dados transmitidos via rede devido a ordem de magnitude associada com o número de métricas.

A documentação oficial do Prometheus recomenda que não se deve exceder mais de 10 pares únicos entre *labels* e todas suas combinações de chave e valor

(https://prometheus.io/docs/practices/instrumentation/#do-not-overuse-labels). Isso pode ser necessário em alguns casos, mas é recomendado que desenvolvedores estejam cientes das implicações já que alta cardinalidade vai afetar a performance das consultas e exigir mais dos bancos de dados de séries temporais. Se o(a) desenvolvedor(a) precisa de informações mais granulares sobre determinados usuários, é recomendado que se explore outras tecnologias como *logs* ou rastros (*tracing*), que serão explicadas mais adiante.

Outra ressalva importante está relacionada à emissão de um grande volume de métricas. Este é um problema comum que deve estar no radar de SREs, times responsáveis pelo monitoramento de serviços e desenvolvedores. Quando uma aplicação envia um número massivo de métricas, os bancos de dados de séries temporais podem suprimir o processamento de métricas de acordo com um limite máximo de requisições permitidas em um determinado período de tempo (*rate limit*). Isso resulta em um monitoramento impreciso, pois os gráficos de *dashboards* e alarmes vão processar apenas uma fração dos *datapoints* que deveriam estar fluindo, o que implica que eventos que impactam a experiência de usuários como erros ou alta latência podem estar passando desapercebidos.

Para tais casos, é recomendado que a emissão de métricas da aplicação seja reavaliada para confirmar se tais métricas precisam mesmo ser emitidas em alta frequência. Ao considerarmos, por exemplo, um servidor HTTP que emite uma métrica para cada requisição, se esta aplicação processa muito tráfego, digamos centenas de milhares de requisições, uma estratégia para este caso é uma agregação da métrica dentro da própria aplicação antes que ela seja emitida. Os desenvolvedores podem implementar uma função que calcule uma média entre os *datapoints* que excederem o número de requisições que podem ser acomodadas pelo banco de dados de séries temporais e emitir estes valores agregados.

Agora que aprendemos mais sobre métricas, vamos entender como as mesmas são utilizadas por aplicações do mundo real. Vamos

começar a explorar um exemplo prático desenvolvendo a nossa própria aplicação, porém, antes botar a mão na massa com o código, vamos nos debruçar sobre uma tecnologia que vem ganhando mais adoção na indústria e que todo SRE deve conhecer: vamos falar um pouco sobre OpenTelemetry.

6.4 OpenTelemetry

OpenTelemetry é um projeto de código aberto que compreende um conjunto de APIs (*Application Programming Interfaces*) e SDKs (*Software Development Kits*) para gerenciar dados de métricas, *logs* e rastros (*traces*) de maneira tecnologicamente agnóstica, ou seja, não opera com ferramentas específicas.

Com OpenTelemetry (ou, às vezes, referenciado apenas como *OTel*), o serviço pode emitir métricas, *traces* e *logs* utilizando a SDK de qualquer linguagem de programação suportada. Esses dados são enviados para uma *OTel pipeline* através de *coletores* (*collectors*) e encaminhados para qualquer destino da escolha do administrador da pipeline através da configuração de exportadores (*exporters*) específicos (por exemplo, *Prometheus exporter* ou *Graphite Exporter*).

Então, como você pode ver, a vantagem da adoção de uma tecnologia como OpenTelemetry é que as aplicações podem utilizar o mesmo código (API/SDK) e a mesma *OTel pipeline* para emitir métricas, *logs* e *traces* para diferentes tipos de bancos de dados de série temporal e soluções de *logs* e rastreamento.

Aqui estão algumas destas tecnologias utilizadas para fins de observabilidade:

- Métricas: Prometheus, Graphite, InfluxDB e TimescaleDB.

- *Logs*: Grafana Loki, Splunk e SolarWinds Observability.
- *Traces*: Jaeger, Zipkin, Grafana Tempo, DataDog e NewRelic.

> *Logs* e *traces* serão discutidos nos próximos capítulos.

Devido à natureza de uma *OTel pipeline*, operadores(as) podem realizar experimentos com diferentes tecnologias e ferramentas sem precisar mudar o código de suas aplicações. Esta estratégia traz muitas vantagens para SREs que desejam realizar migrações entre soluções de monitoramento para buscar melhor observabilidade e fornecer um suporte melhor para os serviços de sua organização.

Para aprendermos mais sobre OTel com exemplos práticos, vamos desenvolver uma aplicação de exemplo para continuar nossos experimentos com métricas.

6.5 Criando uma aplicação de exemplo

Vamos criar uma simples aplicação para responder a requisições HTTP e emitir métricas. Será em Golang, mas os conceitos explicados aqui podem ser encontrados em outras aplicações criadas com outras linguagens de programação.

```
package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/", HelloServer)
    http.ListenAndServe(":8080", nil)
}
func HelloServer(w http.ResponseWriter, r *http.Request) {
```

```
    fmt.Fprintf(w, "Olá, %s!", r.URL.Path[1:])
}
```

Para iniciar o projeto desta aplicação, você deve instalar o Go na sua máquina (https://go.dev/dl/) e seguir os passos descritos a seguir:

1. Crie uma pasta `sreApp`.
2. Dentro da pasta, copie o código acima e cole em um novo arquivo `main.go`.
3. Inicialize um novo módulo para gerenciar suas dependências: `go mod init casadocodigo.com.br/metal-app` (isso será útil depois quando importarmos outros módulos que serão utilizados na aplicação).
4. Execute o programa com: `go run main.go`.

Isso deve iniciar um servidor HTTP localmente, que pode ser acessado em seu browser no endereço: `http://localhost:8080`.

Como você pode inferir pelo código do programa, a mensagem retornada será o texto "Olá, " acompanhando do caminho da URL fornecido pelo cliente. Por exemplo:

```
curl -s "http://localhost:8080/sre"
Olá, sre!
```

Vamos tornar as coisas mais interessantes adicionando algumas linhas de código para inicializarmos um *contador* de requisições utilizando a OpenTelemetry API.

Métricas OTel em ação

O código a seguir contém instruções para inicializar uma aplicação Web com um emissor de métricas, também conhecido como *Medidor* (*Otel Meter*), utilizando os módulos de OpenTelemetry para criar uma métrica simples e ilustrar como gerar telemetria para uma aplicação. Substitua o código da aplicação anterior (`main.go`) pela lógica encontrada a seguir:

```go
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"

    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"
    instrument "go.opentelemetry.io/otel/metric"
    otel_metric "go.opentelemetry.io/otel/metric"
    "go.opentelemetry.io/otel/sdk/metric"
    "go.opentelemetry.io/otel/sdk/resource"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"
)

var meuContador otel_metric.Int64Counter

func main() {
    ctx := context.Background()

    serviceName := "metal-app"
    collectorAddress := "localhost:4317"
    log.Printf("Establishing gRPC connection with %s...\n",
collectorAddress)

    dopts := []grpc.DialOption{
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithBlock(),
        grpc.WithTimeout(5 * time.Second),
    }
    conn, err := grpc.DialContext(ctx, collectorAddress, dopts...)
    if err != nil {
        log.Fatalf("ERROR: Unable to establish grpc connection to otel
```

```
metrics collector: %w", err)
    }

    exporter, err := otlpmetricgrpc.New(ctx,
otlpmetricgrpc.WithGRPCConn(conn))
    if err != nil {
        log.Fatalf("unable to create otel grpc exporter: %w", err)
    }

    var attrs []attribute.KeyValue
    // Always include service_name by default
    attrs = append(attrs, semconv.ServiceName(serviceName))

    res, err := resource.New(ctx, resource.WithAttributes(
        attrs...,
    ))
    if err != nil {
        log.Fatalf("failed to create resource: %w", err)
    }

    provider := metric.NewMeterProvider(
        metric.WithResource(res),
        metric.WithReader(metric.NewPeriodicReader(exporter)),
    )
    meter := provider.Meter("sre")

    c, err := meter.Int64Counter("sre.meu.contador")
    if err != nil {
        log.Fatalf("unable to create counter metric", err)
    }
    meuContador = c

    http.HandleFunc("/", HelloServer)
    http.ListenAndServe(":8080", nil)
}

func HelloServer(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
```

```
    var attrs = []attribute.KeyValue{
        attribute.Key("operacao").String("oi"),
    }
    opt := instrument.WithAttributes(attrs...)

    log.Print("emitindo metrica OTel...")
    meuContador.Add(ctx, 1, opt)

    fmt.Fprintf(w, "Olá, %s!", r.URL.Path[1:])
}
```

O processo de inicialização deste programa envolve uma série de passos, vamos descrevê-los para facilitar o entendimento deste exemplo. Primeiro, declaramos a seguinte variável global:

```
var meuContador otel_metric.Int64Counter
```

Em um programa real, as instâncias de cada métrica devem ser armazenadas em alguma estrutura de dados que possa ser acessada por outras camadas da aplicação. Essa variável foi declarada desta maneira apenas para ilustrar como uma métrica do tipo contador (*counter*) pode ser utilizada.

Logo em seguida, dentro da função `main()`, nós inicializamos um contexto de plano de fundo (`context.Background()`). Este tipo de objeto é particularmente útil em aplicações desenvolvidas com a linguagem Go, pois é passado entre diferentes funções na sequência de operações que compreendem uma determinada funcionalidade da aplicação. O contexto, na linguagem de programação Go, pode ser usado para cancelar alguma tarefa (*cancel*), definir um limite para o tempo de espera da execução de alguma função (*timeout*) ou armazenar informações relacionadas a uma requisição específica (*request-scoped values*).

Depois, nós definimos mais duas variáveis: `serviceName := "metal-app"` e `collectorAddress := "localhost:4317"`, para definir o nome da nossa aplicação e o endereço do *OTel Collector*, respectivamente. Utilizamos o endereço `localhost` pois, mais adiante, nós vamos usar

um contêiner Docker para iniciar uma instância do *OTel Collector* e todos esses processos serão executados localmente. Nas próximas linhas de código, uma lista de parâmetros (`grpc.DialOption`) é definida para configurar uma conexão gRPC (`grpc.DialContext(ctx, collectorAddress, dopts...)`). O gRPC é um protocolo para Chamadas de Procedimento Remoto (*Remote Procedure Call*) cuja conexão, referenciada pela variável `conn` , é utilizada para criar uma instância de um *exportador* de métricas (`exporter`) através do seguinte código: `otlpmetricgrpc.New(ctx, otlpmetricgrpc.WithGRPCConn(conn))` .

Para continuar o nosso experimento de emissão de métricas, devemos primeiro definir as *labels* que serão aplicadas a todas as métricas emitidas nesta aplicação. Tais *labels* comuns a todas as métricas são definidas como um recurso do OTel SDK (*go.opentelemetry.io/otel/sdk/resource*) que é criado através de atributos declarados como uma lista de chave e valor (`var attrs []attribute.KeyValue`) e, a seguir, adicionamos a *label* que corresponde ao nome do serviço `append(attrs, semconv.ServiceName(serviceName))` .

Note que esta *label* é configurada por uma função `ServiceName` do pacote `semconv` . Este pacote contém funções relacionadas a *convenção semântica* (*Semantic Convention*), que representa um conjunto de nomes padronizados para métricas e suas respectivas *labels*, acomodando diferentes tipos de operações e dados. Por exemplo, `system.cpu.utilization` e `http.server.request.duration` são métricas para monitorar a utilização do CPU e a duração de uma requisição HTTP, respectivamente. Esses nomes de métricas com esta notação onde os fragmentos de texto são separados por um ponto (.), ao serem associados com métricas OTel que são coletadas e exportadas para o Prometheus, são convertidas para a notação com *underscores*, exemplo: `system_cpu_utilization` ou `http_server_request.duration` .

A *convenção semântica* é muito útil para padronizar métricas e tornar automações, alarmes, *dashboards* e toda a prática de monitoramento muito mais intuitiva e eficiente. Você pode encontrar

mais detalhes sobre isso na documentação oficial: https://opentelemetry.io/docs/specs/semconv/general/metrics/.

Após definirmos estas configurações iniciais, procedemos com a criação de um *Provedor de Medidores* (*MeterProvider*) e um *Medidor* (*Meter*):

```
provider := metric.NewMeterProvider(
    metric.WithResource(res),
    metric.WithReader(metric.NewPeriodicReader(exporter)),
)
meter := provider.Meter("sre")
```

Repare que o *Meter Provider* é criado com a instância de recurso (`res`) que contém a label `service_name` (de acordo com a *convenção semântica*) e com o exportador (`exporter`). Dessa forma, toda métrica criada por um *Meter* que seja fornecido por este *MeterProvider* vai possuir a mesma *label* com o mesmo valor e seus *datapoints* serão exportados para o *Collector* definido no mesmo exportador. Com a instância do *MeterProvider*, nós utilizamos uma função `Meter()` para obtermos um *medidor*: `provider.Meter("sre")` . Cada medidor deve ter um nome único então utilizamos o nome "sre" arbitrariamente.

Com o nosso *Meter* inicializado, podemos finalmente criar a nossa métrica:

```
c, err := meter.Int64Counter("sre.meu.contador")
if err != nil {
    log.Fatalf("unable to create counter metric", err)
}
meuContador = c
```

Utilizando a função `Int64Counter()` , criamos uma métrica do tipo contador (*counter*) com o nome `sre.meu_contador` e, após a checagem de erro (`if err != nil`), associamos a instância do contador que foi criado à variável global `meuContador` (definida no início do programa), apenas para que possamos acessá-la em outra função logo depois.

Após criarmos nossa métrica, vamos iniciar o servidor HTTP da aplicação:

```
http.HandleFunc("/", HelloServer)
http.ListenAndServe(":8080", nil)
```

A instrução `http.HandleFunc()` permite que o desenvolvedor especifique qual função deve processar as requisições HTTP de acordo com um determinado caminho de URL. Como o nosso programa é bem simples, temos apenas uma operação em nossa API (`HelloServer`), que está mapeada no caminho `/` . Na linha seguinte, vamos iniciar o servidor HTTP com `http.ListenAndServe()` . Esta outra função vai fazer com que o programa fique "escutando" e "servindo" requisições HTTP e, enquanto isso, o *Medidor* vai exportar qualquer métrica que possuir novos *datapoints*, realizando a emissão de métricas a cada 60 segundos.

Agora que explicamos todas as instruções encontradas na função `main()` , vamos, finalmente, explorar o código dentro da função `func HelloServer(w http.ResponseWriter, r *http.Request)` , que vai criar os *datapoints* da nossa métrica.

```
ctx := r.Context()

var attrs = []attribute.KeyValue{
    attribute.Key("operacao").String("oi"),
}
opt := instrument.WithAttributes(attrs...)
```

Em suas primeiras linhas de código, a função `HelloServer()` obtém uma instância do contexto de cada requisição utilizando o objeto *request* que é populado internamente pelo módulo HTTP através da função `ctx := r.Context()` . Isso será bastante útil em um capítulo adiante onde vamos explorar rastros (*traces*).

Logo depois inicializamos uma lista de atributos (`go.opentelemetry.io/otel/attribute`), que serão convertidos em *labels* quando esta métrica for exportada para o Prometheus. O código

define uma *label* com a chave "operacao" e o valor "oi" (`attribute.Key("operacao").String("oi")`), apenas para ilustrar como são definidas uma ou mais *labels* para uma métrica. A lista de atributos é utilizada para popular uma variável `opt` (do tipo `go.opentelemetry.io/otel/metric".MeasurementOption`) que, por sua vez, é passada como um argumento para a próxima função.

```
log.Print("emitindo metrica OTel...")
meuContador.Add(ctx, 1, opt)

fmt.Fprintf(w, "Olá, %s!", r.URL.Path[1:])
```

Para cada requisição HTTP processada por esta aplicação, a função vai utilizar a variável global `meuContador` para registrar uma mudança em nossa métrica. O contador criado na função `main()` através do código `meter.Int64Counter("sre.meu.contador")` vai resultar em uma métrica em formato Prometheus com o nome `sre_meu_contador_total` . O nome da métrica é originalmente definido com pontos (.) que delimitam cada palavra do nome da métrica e, uma vez que a métrica é exportada pelo *Collector* em formato compatível com Prometheus, cada ponto é convertido para *underscore* (_) e, como esta é uma métrica do tipo contador, o sufixo `_total` será adicionado ao nome da métrica.

Agora que exploramos o código da nossa aplicação de exemplo, estamos com quase tudo pronto para inicializarmos os contêineres e executar o programa para vermos a nossa métrica em ação. Porém, antes de proceder, devemos introduzir as configurações necessárias para o *Collector* e a extração de métricas do Prometheus.

Para inicializarmos o *Collector*, precisamos de um arquivo de configuração `collector-config.yaml` , que deve conter o seguinte código:

```
receivers:
  otlp:
    protocols:
```

```
      grpc:
      http:

exporters:
  prometheus:
    endpoint: "0.0.0.0:9464"

processors:
  batch:

service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [prometheus]
      processors: [batch]
```

Esta configuração basicamente define os receptores (*receivers*), processadores (*processors*) e exportadores (*exporters*) da *Otel pipeline* que vai entrar em operação dentro do contêiner que abriga o *Collector*. Repare que existe um *receiver* definido para o protocolo *gRPC* e um *exporter* do tipo Prometheus cuja configuração contém um endereço `0.0.0.0:9464`, que é o endereço onde nossa métrica será encontrada.

A configuração do Prometheus deve ser modificada para extrair as métricas expostas pelo *Collector*, portanto, o código do arquivo `prometheus.yml` precisa de uma nova instrução no bloco `scrape_configs`:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: "extraindo-metricas-do-grafana"
    static_configs:
      - targets: ["grafana:3000"]
  - job_name: "extraindo-metricas-do-otel-collector"
```

```
    static_configs:
      - targets: ["collector:9464"]
```

Repare que o novo *job* de extração de métrica aponta para o contêiner com o nome *collector* e a porta 9464, que corresponde à configuração do Prometheus exporter definida no `collector-config.yaml`).

Vamos dar seguimento ao nosso experimento e inicializar os contêineres. Para iniciar o contêiner `prometheus` com a nova configuração, interrompa o contêiner que foi inicializado anteriormente:

```
docker stop prometheus
```

Logo em seguida, certifique-se de que você está no mesmo diretório em que os arquivos de configuração foram criados e inicie o contêiner Prometheus novamente.

```
docker run --rm -d --name prometheus -p 9090:9090 --network sre-test-network -v $(PWD)/prometheus.yml:/etc/prometheus/prometheus.yml --entrypoint sh prom/prometheus -c "/bin/prometheus --config.file=/etc/prometheus/prometheus.yml"
```

E aqui está o comando para iniciar o contêiner do OTel *Collector*:

```
docker run --rm -d --name=collector -p 4317:4317 -p 9464:9464 -p 55680:55680 -p 55681:55681 -v $(PWD)/collector-config.yaml:/conf/collector-config.yaml --network sre-test-network otel/opentelemetry-collector-contrib:latest --config /conf/collector-config.yaml
```

Finalmente estamos prontos para executar a nossa aplicação novamente para observarmos a nossa métrica em ação. Para realizar o download dos módulos necessários para executar a aplicação (de acordo com as dependências declaradas no `block` de `import`), utilize o comando `go mod tidy` e, logo em seguida, execute a aplicação com `go run main.go`.

Para interagir com o servidor HTTP da aplicação, envie uma requisição visitando o endereço `http://localhost:8080/sre` ou com um simples comando `curl` :

```
curl -s "http://localhost:8080/sre"
```

Isso deve produzir *logs* contendo mensagens semelhantes ao exemplo abaixo:

```
2024/05/18 12:46:15 emitindo metrica OTel...
```

Para enviar múltiplas requisições com um único comando, você pode utilizar instruções de *Bash scripting* (linguagem de scripts encontrada em terminais Linux):

```
for i in {1..5}; do curl -s "http://localhost:8080/sre"; done
```

Estes comandos devem registrar mudanças no contador, cuja métrica pode ser inspecionada no endereço `http://0.0.0.0:9464/metrics` .

> O Medidor (*Meter*) do OpenTelemetry emite métricas a cada 60 segundos por padrão, então você deve aguardar um minutinho até que a métrica apareça. Este é o alvo da extração de métricas do Prometheus, portanto, se a métrica não for encontrada ao visitar o endereço, alguma coisa está errada e você deve tentar retomar os passos e verificar os *logs* dos contêineres com o comando `docker logs <container_id>` .

Você deve confirmar a presença da métrica em formato Prometheus exibindo o número de vezes em que a nossa aplicação recebeu uma requisição HTTP, o resultado deve ser similar a:

```
# HELP sre_meu_contador_total
# TYPE sre_meu_contador_total counter
sre_meu_contador_total{job="metal-app",operacao="oi"} 20
```

Para criar um gráfico com esta métrica, visite o endereço `http://localhost:9090/graph` , clique na aba *graph* e execute a seguinte consulta com *PromQL*: `sum(rate(sre_meu_contador_total[5m]))` . Envie mais requisições HTTP para `"http://localhost:8080/sre"` e veja como a linha se comporta no gráfico. Aqui está um exemplo:

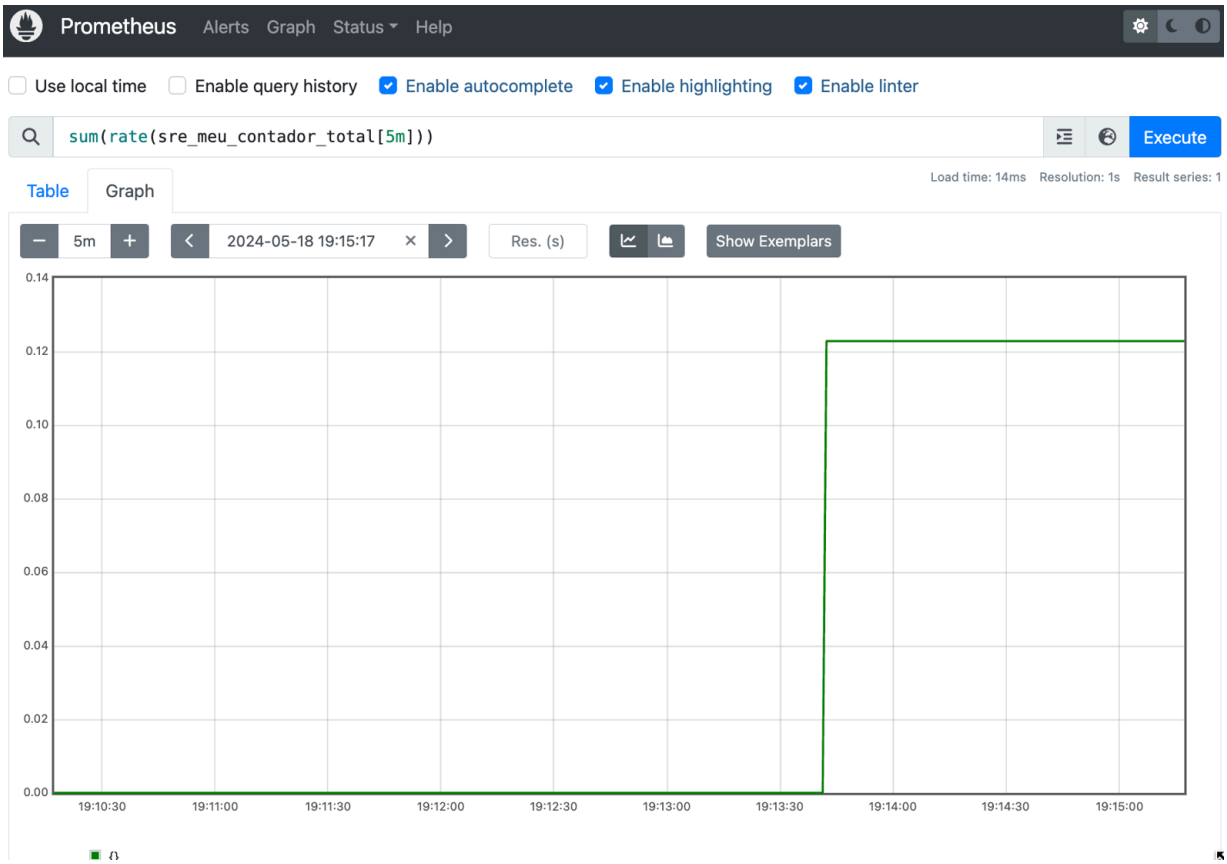

Figura 6.6: Criando gráficos com uma métrica customizada

O Prometheus possui sua própria interface Web para realizar consultas e criar gráficos a partir de métricas, porém, como um serviço geralmente emite múltiplas métricas de diferentes fontes de dados, operadores(as) decidem observá-las em um *dashboard*, que compreende uma única tela com vários painéis, fornecendo uma visualização da telemetria de uma maneira mais eficiente. Existem várias ferramentas de monitoramento que oferecem *dashboards* entre suas funcionalidades, como DataDog ou NewRelic, mas, para

facilitar nosso experimento, vamos utilizar a própria aplicação Grafana, que é bastante poderosa e tem suporte para várias fontes de dados (em inglês: *data sources*).

Para criar um *data source* em Grafana, você deve navegar para `http://localhost:3000/datasources` (não esqueça do login e senha padrão: `admin/admin`), clique em *Add new data source*, selecione Prometheus na seção de *Time series databases* e, quando alcançar a página de criação de um novo *data source* do tipo Prometheus, apenas preencha o campo do formulário *Prometheus server URL* com o valor `http://prometheus:9090` e clique no botão *Save & test* que se encontra no fim da página. Aqui está uma imagem para ilustrar este passo:

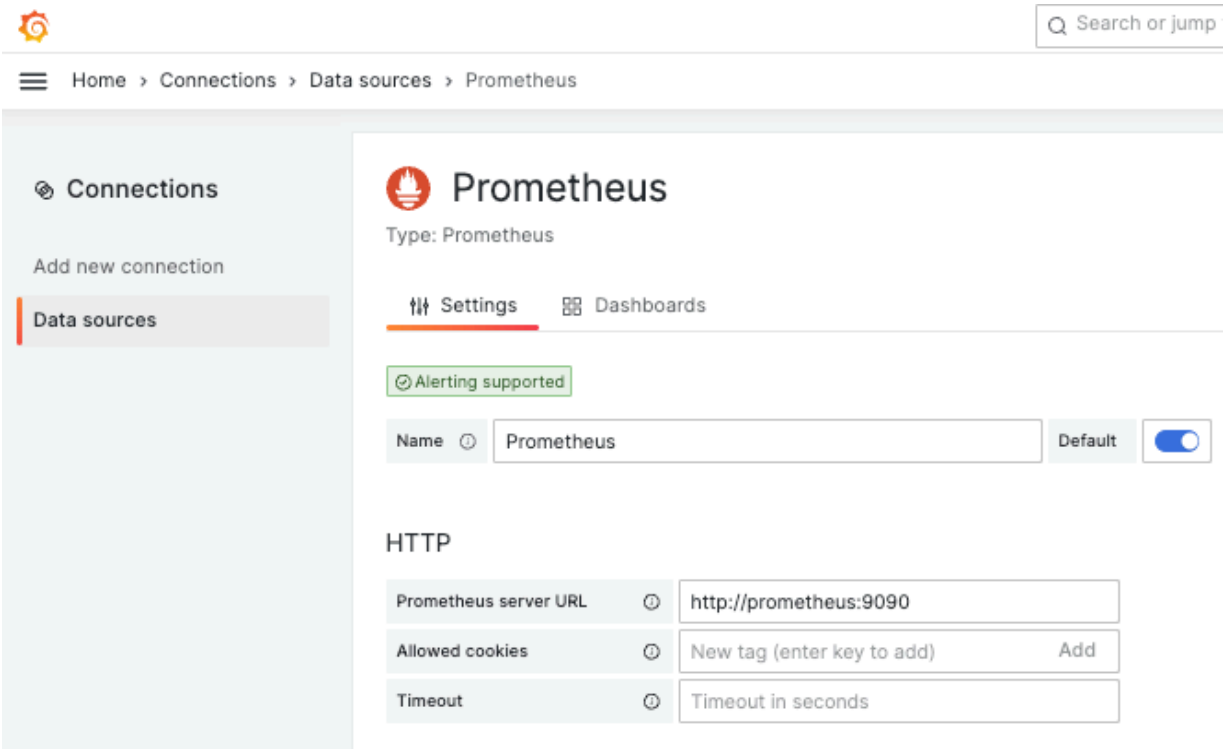

Figura 6.7: Criando um data source do tipo Prometheus em Grafana

Agora que temos um *data source*, podemos navegar para `http://localhost:3000/dashboards` e clicar em *New* e, logo depois, na opção *New Dashboard*. Clique em *Add visualization* para criar o primeiro painel deste dashboard, o que deve trazer uma janela de

seleção de *data sources*. Escolha *Prometheus* e prossiga com o restante da configuração do painel. Logo abaixo do painel, na aba *Query*, encontra-se a seção onde operadores(as) podem adicionar consultas, então copie e cole a instrução *PromQL* `sum(rate(sre_meu_contador_total[5m]))` no campo de texto e execute a consulta clicando em *Run queries*. Esta ação deve renderizar o gráfico que corresponde a sua métrica no painel e você deverá observar algo similar à imagem abaixo:

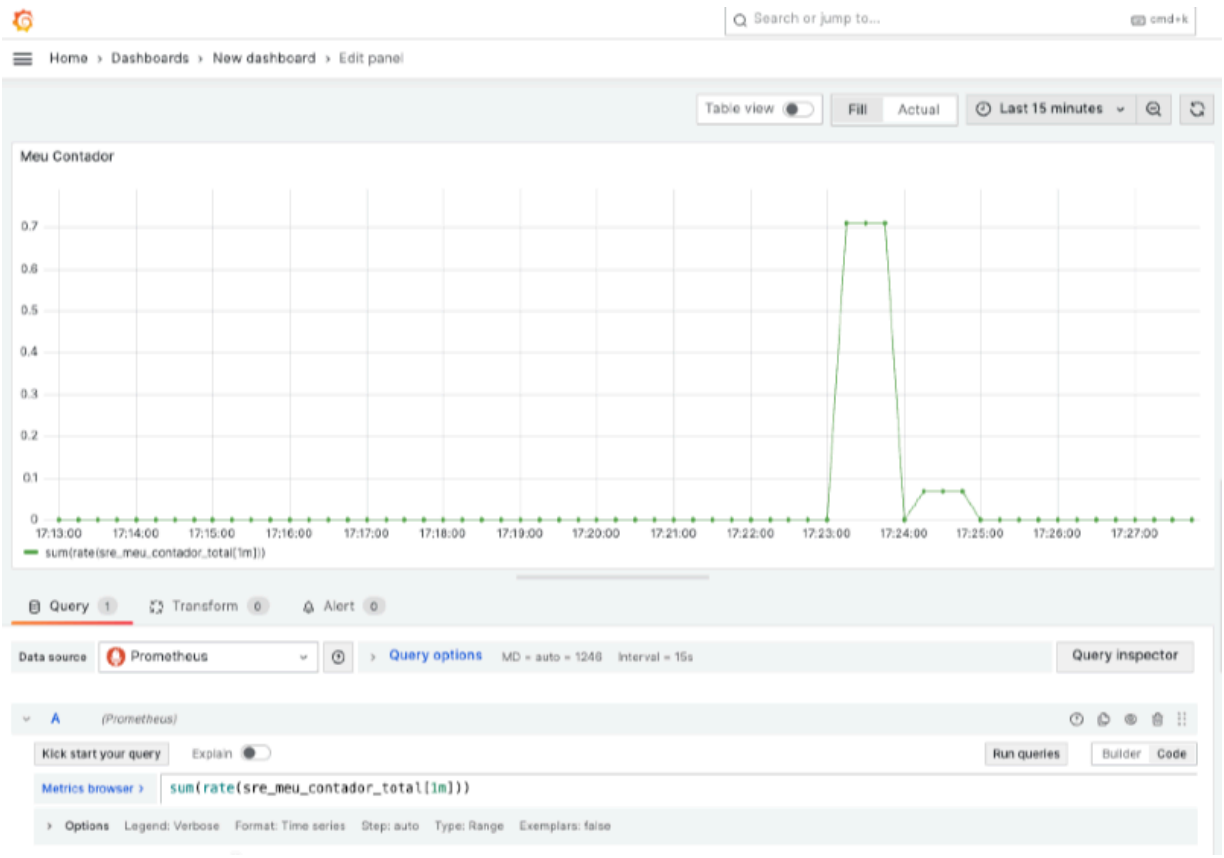

Figura 6.8: Configurando um painel para criar um gráfico a partir de uma métrica prometheus

Clique no botão *save* no canto superior direito da tela e forneça um nome para o seu *dashboard*. Se todos os passos foram seguidos corretamente, você terá criado um dashboard com um painel contendo um gráfico gerado a partir dos *datapoints* de sua métrica customizada. Envie mais requisições para a aplicação de exemplo (`"http://localhost:8080/sre"`), espere até que as métricas sejam

emitidas para o Otel *Collector*, clique no botão *Refresh dashboard* que se encontra no canto superior direito de seu *dashboard* e veja novamente como gráfico reage aos novos *datapoints*.

Como mencionado anteriormente, as métricas são emitidas a cada 60 segundos por padrão, porém, se você deseja observar os resultados mais rapidamente, você pode adicionar o código abaixo à função `main()` (antes da chamada `http.ListenAndServe`) para criar um *loop* que vai forçar a emissão das métricas em uma frequência de tempo mais curta:

```
metricEmissionTicker := time.NewTicker(5 * time.Second)

go func() {
    for range metricEmissionTicker.C {
        log.Print("emitting metrics...")
        if errMeter := provider.ForceFlush(ctx); errMeter != nil {
            log.Print("otel metric emitter flush failure")
            log.Print(errMeter)
        }
    }
}()
```

Neste capítulo nós conduzimos nosso primeiro experimento com uma métrica customizada criada "do zero". Este exemplo deve servir como um fundamento para explorar a API e a SDK do OpenTelemetry e criar métricas de acordo com as necessidades da organização, seja para fins de manutenção de serviços, criação de alarmes ou para gerar indicadores que ajudem a avaliar o sucesso de um produto.

Capítulo 7
Logs

Logs são utilizados para registrar eventos de maneira mais descritiva, já que autores(as) do código podem definir qualquer mensagem e concatenar uma ou mais variáveis para que seja possível inspecionar as operações de um sistema e os dados que estão sendo manipulados em um espaço de tempo específico. Portanto, *logs* são recursos vitais em uma eventual investigação que se torna necessária caso o serviço apresente problemas ou durante o desenvolvimento para averiguar se o sistema está funcionando como esperado. Devido à sua habilidade de imprimir uma grande quantidade de detalhes, *logs* são utilizados para análises mais profundas que buscam a causa raiz de falhas em uma aplicação e, se formatados de maneira adequada, podem ser utilizados para fins de análises de dados e até mesmo para a geração de gráficos.

Logs são bastante úteis para SREs, especialmente quando precisam investigar a causa raiz de um incidente ou analisar alguma anomalia no sistema. Entradas de *logs* como esta encontrada a seguir, que fazem parte de um arquivo liberado ao público contendo *logs* de acesso do servidor Web do *NASA Kennedy Space Center*, apresentam muitas informações que podem ser usadas para entender que tipo de atividades estavam em curso no website naquela data e hora.

```
ad18-037.compuserve.com - - [07/Aug/1995:15:35:53 -0400] "GET
/shuttle/missions/sts-71/images/KSC-95EC-0917.jpg HTTP/1.0" 200 52491
144.13.1.208 - - [07/Aug/1995:15:35:54 -0400] "GET
/shuttle/countdown/countdown.html HTTP/1.0" 200 4673
```

A primeira coluna apresenta o endereço do cliente que enviou a requisição (`ad18-037.compuserve.com`). Logo depois, temos um bloco de informações contendo data, hora e o fuso horário, também conhecido como *timestamp* (`[07/Aug/1995:15:35:53 -0400]`), seguido

de mais informações relacionadas à requisição HTTP como o verbo (`GET`), a URL (`/shuttle/missions/sts-71/images/KSC-95EC-0917.jpg`) e a versão do protocolo (`HTTP/1.0`). E, nas duas últimas colunas, o código de resposta HTTP (`200`) e o número de bytes na resposta (`52491`). Para mais informações sobre a estrutura de *logs* de acesso do Apache HTTP Server, visite https://httpd.apache.org/docs/2.4/logs.html.

Ao analisar *logs*, SREs podem correlacionar eventos com requisições originadas de um mesmo cliente, ou que têm como alvo os mesmos recursos (URLs), que possuem códigos de resposta inesperados (como `400` ou `500` , por exemplo) ou mapear requisições com seus *timestamps* e correlacionar com outros eventos que impactam o sistema.

Outro exemplo dos benefícios de *logs* para a investigação de problemas está na presença de informações detalhadas relacionadas a problemas que ocorrem em tempo de execução, como as *stack traces*. A mensagem de *log* a seguir ilustra um erro bastante comum que ocorre quando o programa tenta atribuir um valor a um mapa que não foi inicializado (*nil pointer*):

```
panic: assignment to entry in nil map

goroutine 1 [running]:
casadocodigo.com.br/livro-app/internal/negocio.ExecutarFuncionalidade(...)
    /Users/myuser/go-workspace/livro-
app/internal/negocio/funcionalidade.go:6
casadocodigo.com.br/livro-app/internal/api.
(*LivroAppAPI).RealizarTarefa(...)
    /Users/myuser/go-workspace/livro-app/internal/api/app.go:10
main.main()
    /Users/myuser/go-workspace/livro-app/cmd/main.go:11 +0x3c
exit status 2
```

Repare que a *stack trace* contém a sequência de chamadas em cada função, exibindo a linha de código exata em que o pânico ocorreu, assim como os respectivos nomes de cada script, seus pacotes e o

caminho para os arquivos. Para análises que exigem este nível de detalhe, os *logs* são recursos importantes que devem estar à disposição para SREs em situações de triagem de incidentes ou *Post Mortems*.

7.1 Logs em ação

Para explorarmos a utilização de *logs*, vamos modificar a nossa aplicação de exemplo mais uma vez. Vamos começar inserindo algumas linhas de código para criar um arquivo de log. Este arquivo será escaneado pelo *OTel Collector* para que cada entrada do *log* seja exportada para o *Loki*, que é a ferramenta *open source* para armazenamento e consulta de *logs* do GrafanaLabs (https://grafana.com/oss/loki/).

Vamos ao código. Primeiro, certifique-se de que os seguintes módulos estão importados no início de nosso script. Devemos adicionar o módulo `os` para interagir com funções do sistema operacional e inserir *logs* em um arquivo no disco local:

```
import (
    "context"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
```

A lógica a seguir deve criar um arquivo no diretório raiz da nossa aplicação e declarar uma variável global `logger` para referenciar uma instância do objeto *Logger*, que é inicializado através da função `New` (mais detalhes em https://pkg.go.dev/log#New):

```
var (
    outfile, _ = os.Create("minhaApp.log")
```

```
    logger     = log.New(outfile, "", 0)
)
```

Agora nós devemos substituir as linhas de código contendo `log.Printf`, que são aquelas que imprimem mensagens na saída padrão do terminal (*standard output*), por funções `logger.Printf` para utilizarmos o novo gerador de *logs* (`logger`) que está configurado para escrever as mensagens no arquivo local. Aqui estão as duas instruções que utilizavam `log.Printf` na versão anterior da nossa aplicação devidamente substituídas:

```
    logger.Printf("Establishing gRPC connection with %s...\n",
collectorAddress)
...
    logger.Print("emitindo metrica OTel...")
```

Execute o programa e perceba que o terminal não exibe mais nenhuma mensagem, inicie um novo terminal e verifique o conteúdo do arquivo `minhaApp.log`. Você pode utilizar o comando linux `tail` com a opção `-f` para seguir (*follow*) as novas linhas de *log* que surgirem no arquivo (`tail -f minhaApp.log`). Você deve encontrar as mesmas mensagens que surgiam ao executar a versão anterior do programa:

```
% tail -f ~/go-workspace/metal-app/emissor-de-metricas-
simples/minhaApp.log
Establishing gRPC connection with localhost:4317...
emitindo metrica OTel...
emitindo metrica OTel...
emitindo metrica OTel...
```

No próximo passo, nós vamos modificar a configuração do nosso OTel *Collector* (`collector-config.yaml`) para que ele possa consumir o arquivo de log. Para que estes *logs* sejam recebidos, processados e exportados pela *pipeline* do nosso OpenTelemetry Collector temos que atualizar a configuração do OTel *Collector* com as seguintes instruções:

```yaml
receivers:
  otlp:
    protocols:
      grpc:
      http:

  filelog:
    include: [ /var/log/sre/minhaApp.log ]

exporters:
  prometheus:
    endpoint: "0.0.0.0:9464"
  loki:
    endpoint: http://loki:3100/loki/api/v1/push
    default_labels_enabled:
      exporter: true
      job: true

processors:
  batch:
  resource:
    attributes:
    - key: service_name
      value: "metal-app"
      action: upsert
    - key: deployment_environment
      value: "dev"
      action: upsert
    - action: insert
      key: loki.resource.labels
      value: service_name, deployment_environment

service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [prometheus]
      processors: [batch]
    logs:
```

```
      receivers: [filelog]
      processors: [resource]
      exporters: [loki]
```

Os novos parâmetros dessa configuração compreendem:

- O receptor (*receiver*) `filelog` : para definir uma lista de caminhos de arquivos cujas entradas devem ser consumidas pelo coletor.
- Um novo exportador (*exporter*): `loki` , que será a destino dos *logs* produzidos pela aplicação (mais detalhes a seguir).
- Outro processador (*processor*) chamado `resource` , onde serão definidos mais metadados (*labels*) para as entradas de *logs* antes de sua exportação.
- Uma nova *pipeline* para `logs` contendo os respectivos *receivers*, *processors* e *exporters* mencionados anteriormente.

Antes de testar nosso OTel *Collector* com estas novas configurações, devemos inicializar um contêiner Loki para receber os *logs* através do endereço `http://loki:3100/loki/api/v1/push` (de acordo com a configuração do *Collector*).

Para configurar uma instância do Loki localmente, execute o seguinte comando:

```
docker run --rm -d --name=loki --mount source=loki-data,target=/loki -p
3100:3100 --network sre-test-network grafana/loki
```

Interrompa o OTel *Collector* que está em operação com `docker stop collector` e reinicie o contêiner com a nova configuração:

```
docker run --rm -d --name=collector -p 4317:4317 -p 9464:9464 -p
55680:55680 -p 55681:55681 -v $(PWD)/collector-
config.yaml:/conf/collector-config.yaml -v $(PWD)/emissor-de-metricas-
simples/minhaApp.log:/var/log/sre/minhaApp.log --network sre-test-network
otel/opentelemetry-collector-contrib:latest --config /conf/collector-
config.yaml
```

Repare que existe uma instrução adicional para *montar* um volume neste contêiner `-v $(PWD)/emissor-de-metricas-simples/minhaApp.log:/var/log/sre/minhaApp.log`, e o caminho do arquivo de *log* corresponde à configuração do receptor (*receiver*) que está declarado em nosso arquivo `collector-config.yaml`. O arquivo de *log* será lido dentro do contêiner do nosso *Collector* para que possa ser processado e exportado de acordo com a *pipeline* que foi configurada.

Para enviar *logs* através da *pipeline*, a aplicação deve produzir *logs*, portanto você pode enviar requisições para a aplicação (`curl -s "http://localhost:8080/sre"`) ou adicionar mais mensagens ao arquivo de *log* com um comando de *Bash scripting* como:

```
echo "testando testando" >> ~/<caminho_do_projeto>/minhaApp.log
```

Seguindo passos similares ao do capítulo anterior, vamos adicionar um *data source* do tipo Loki no console de administração Grafana para executarmos consultas e visualizarmos *logs* no console Web.

Navegue novamente para `http://localhost:3000/datasources`, clique em *Add new data source*, selecione a opção *Loki* e, finalmente, na tela de criação de *data source*, preencha o campo *URL* com o endereço `http://loki:3100` e clique em *Save & test* no fim da página. A imagem a seguir mostra como criar tal *data source*:

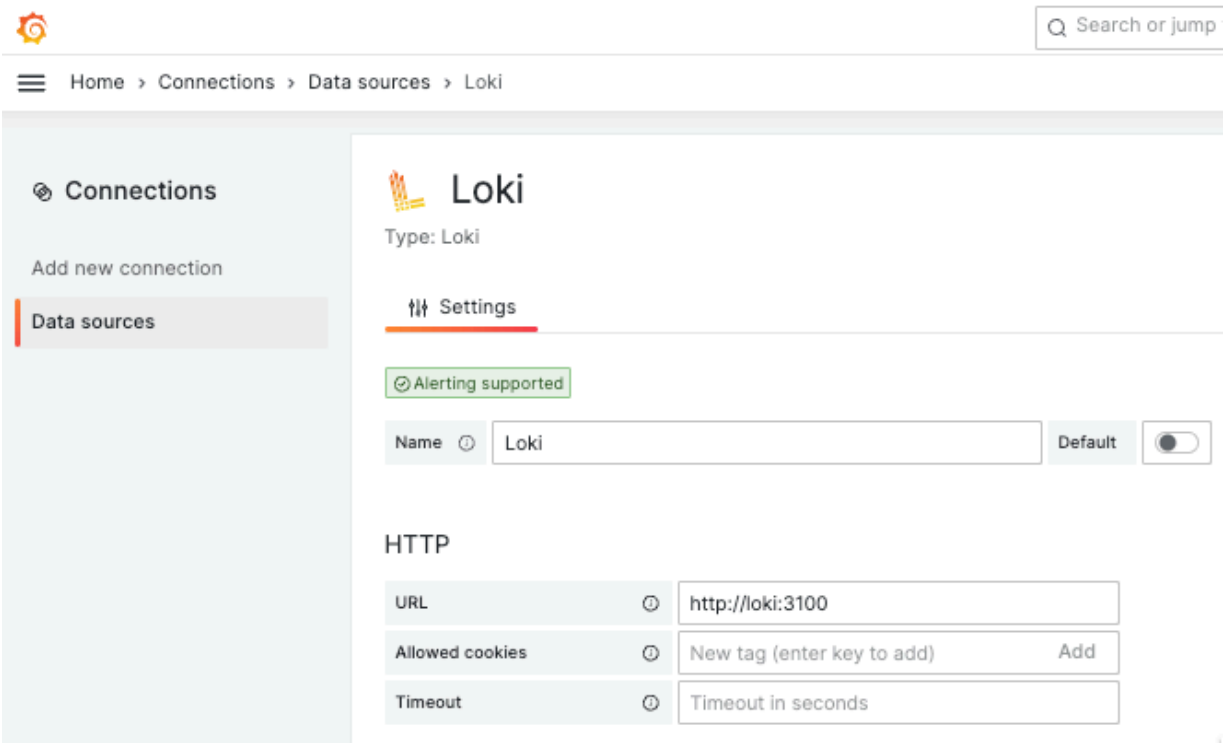

Figura 7.1: Criando um data source do tipo Loki em Grafana

Se tudo estiver configurado corretamente, podemos consultar os *logs* da aplicação no console `http://localhost:3000/explore`. Ao navegar para esta página e selecionar o *data source* Loki criado recentemente, você pode utilizar a interface de consulta do tipo *builder* e escolher *labels* através dos campos de seleção abaixo de *label filters*, ou mudar para a aba *code* cuja seleção se encontra no lado direito da tela e inserir a seguinte consulta em formato LogQL (*Log Query Language*):

```
{service_name="metal-app",deployment_environment="dev"}
```

Ao clicar em *Run query* no canto superior direito da tela (ou você pode segurar *Shift* e pressionar *Enter* para executar consultas), as entradas de *logs* devem surgir logo abaixo do campo de consulta. Cada registro de *log* deve incluir a mensagem produzida pela aplicação e estará acompanhado de seu respectivo *timestamp* (data & hora). A imagem a seguir mostra uma consulta de LogQL em funcionamento:

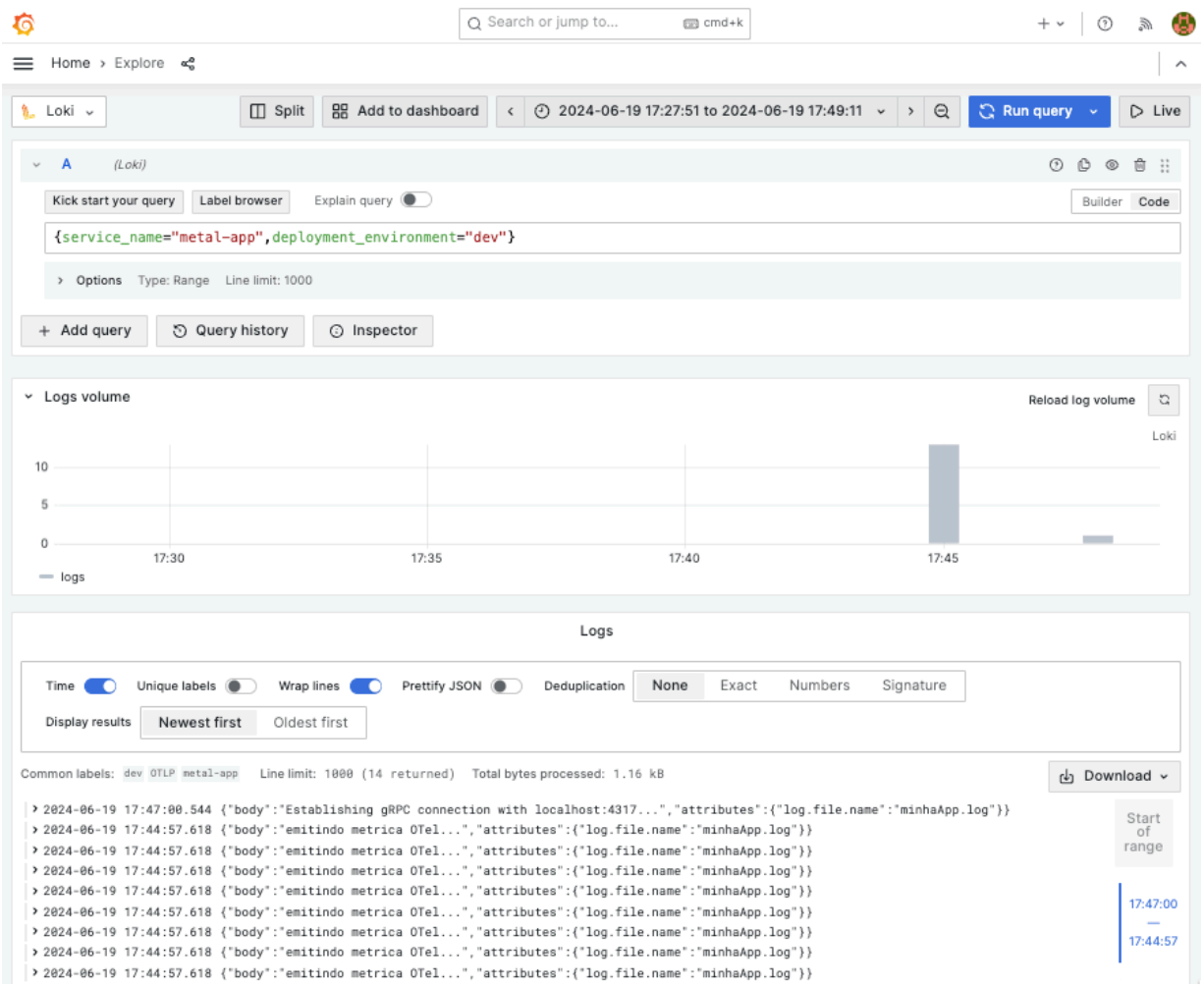

Figura 7.2: Executando uma consulta de logs com LogQL

LogQL é uma ferramenta bem poderosa que permite que operadores(as) filtrem *logs* e criem gráficos a partir de entradas de *logs*. Vamos explorar um exemplo prático simulando um erro em nossa aplicação para que possamos depois buscar as mensagens de erro através de consultas ao Loki. Para simplificar esse experimento vamos injetar uma entrada de *log* em nosso arquivo que está sendo consumido pelo OTel *Collector*:

```
echo "ERRO: alguma coisa deu errado" >>
~/<caminho_do_projeto>/minhaApp.log
```

Agora, na interface Web, vamos tentar a seguinte consulta:

```
{service_name="metal-app",deployment_environment="dev"} |= "ERRO"
```

Repare que apenas as entradas de *log* que contêm o texto "ERRO" vão surgir no resultado da busca. Você pode utilizar também expressões regulares para não se preocupar com letras maiúsculas ou minúsculas (Por exemplo: `|~ "(?i)erro.*"`), ou até mesmo para construir filtros mais complexos. A consulta também pode possuir um filtro negativo `!= "ERRO"` , para retornar apenas as entradas de *log* que não possuem o texto especificado.

Para criar gráficos, seguindo este exemplo do filtro de erros, nós podemos utilizar a seguinte consulta `sum(rate({service_name="metal-app",deployment_environment="dev"} |~ "(?i)erro.*"[5m]))` , para vermos a taxa de mudança no número de *logs* produzidos nos últimos 30 minutos, o que pode ser bastante útil para correlacionar algum evento problemático em outros pontos do sistema. Aqui está uma imagem ilustrando esta consulta:

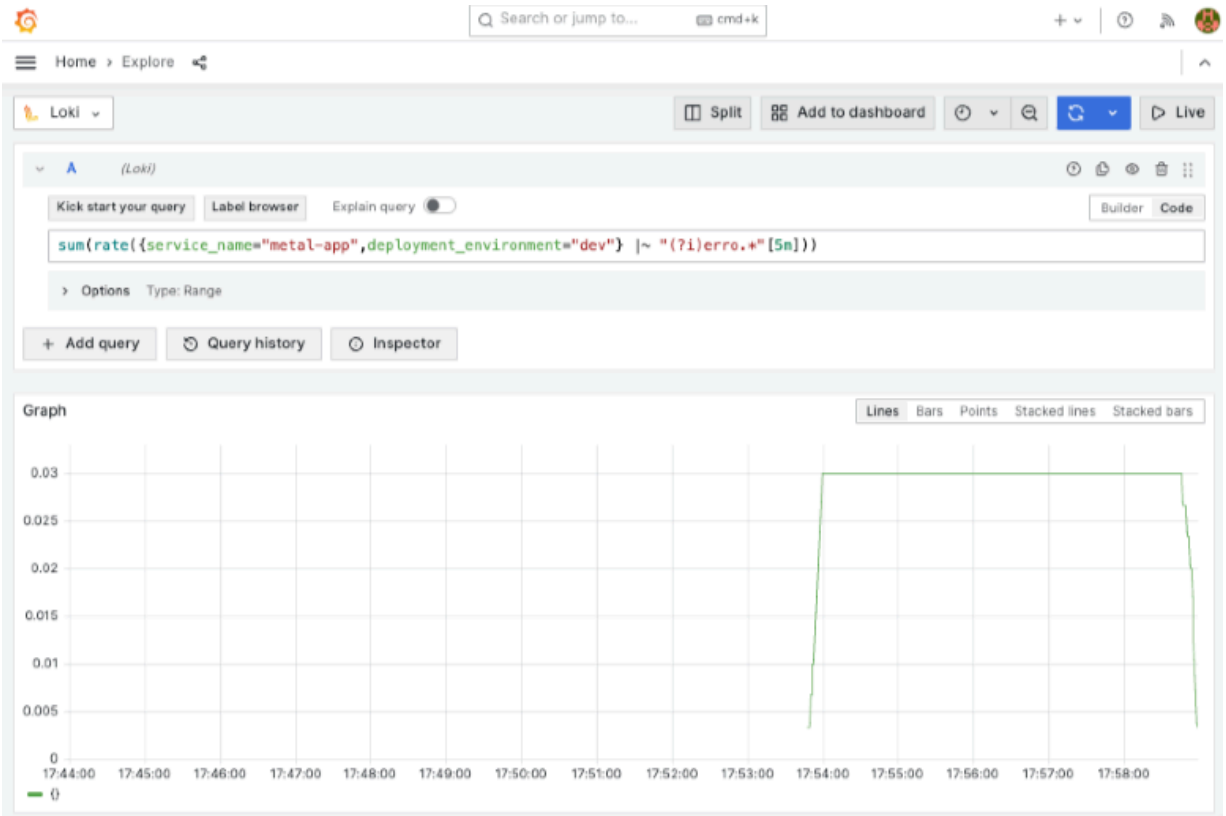

Figura 7.3: Criando um gráfico através de uma consulta LogQL

Como você pode ver, o formato de tais consultas é bastante similar ao PromQL (linguagem de consulta do Prometheus). Mais informações sobre LogQL e sua sintaxe podem ser encontradas na documentação oficial: https://grafana.com/docs/loki/latest/query/.

Com mais um experimento concluído, encerramos o nosso capítulo sobre *logs*. A seguir, vamos explorar o rastreamento de aplicações, também conhecido como *traces*.

Capítulo 8
Traces

Os rastros (em inglês: *traces*) são cadeias de eventos que nos ajudam a visualizar como uma transação percorre diferentes pontos de um sistema. Os *traces* são usados no monitoramento de um sistema distribuído para fins de análise da causa raiz de algum problema, mapear dependências ou otimização de performance.

Vamos considerar, por exemplo, uma requisição HTTP que é iniciada pelo website de uma loja virtual para realizar uma compra: se o código da aplicação que recebe a requisição possuir uma configuração de rastreamento (*tracing*), então os operadores(as) podem utilizar os *traces* para observar a jornada que a transação inicia em uma aplicação que processa um pedido, que depois se comunica com outras aplicações para processar o pagamento, atualizar o estoque, gerar recibo, enviar emails etc. Dessa forma, operadores(as) podem observar os rastros de cada requisição através de diferentes pontos de um sistema.

Os rastros apresentam detalhes sobre as operações realizadas dentro de um ou mais serviços, mostrando diferentes funções que são executadas e requisições enviadas a outros serviços. Aqui está uma amostra de como um *trace* pode ser apresentado:

```
aplicativo-backend-X GET servico-busca-produtos /products/{} (300ms)
  aplicativo-backend-X  HTTP GET (100ms)
    servico-busca-produtos /products/id (50ms)
      servico-busca-produtos funcs.GetProduct (50ms)
        servico-busca-produtos funcs.GetProductFromDatabase (100ms)
```

A primeira linha representa toda a operação com o respectivo tempo total de sua execução. Depois vemos cada passo executado em outras funções do mesmo serviço (`aplicativo-backend-X`) e, logo abaixo, outro serviço (`servico-busca-produtos`) e suas respectivas funções informando o tempo gasto em cada uma destas operações

(totalizando os `300ms` como descrito no início). A jornada completa de uma transação é considerada um *trace* e cada ponto em que uma operação específica é executada é considerado um *span*.

Traces são extremamente úteis para analisar as "entranhas" de um sistema, já que eliminam a necessidade de possuir um conhecimento atualizado da arquitetura de toda uma solução e não exige a carga cognitiva de memorizar a sequência de chamadas (por exemplo: "serviço A manda requisições para serviço B, que, por sua vez, manda requisições para serviço C"). *Traces* nos ajudam a observar cada requisição de um cliente de maneira isolada, e cada *span* de um *trace* possui uma série de atributos como `http.client_ip`, `http.host`, `http.method` e `http.route` (e qualquer outra informação que possa ser incluída em cada *span* através do código de sua aplicação), o que é incrivelmente útil para triagens e resolução de problemas (*troubleshooting*).

Entretanto, devido à alta dimensionalidade de *traces*, não é viável processar e armazenar *traces* e *spans* de cada requisição singularmente se o serviço recebe milhares ou centenas de milhares de requisições em um curto espaço de tempo (esse processamento e armazenamento de dados pode ficar bastante caro). É por isso que *traces* geralmente são configurados com uma amostragem (*sampling*) em sistemas de alta vazão (*high throughput*), e a análise de apenas uma porção das requisições que viajam através de diferentes serviços torna-se suficiente para observar o sistema como um todo.

Como funcionam os Traces

O processo de rastreamento funciona da seguinte forma: quando uma requisição é iniciada, um identificador `trace-id` é gerado para aquele rastro específico; logo depois, o mesmo identificador é aplicado em cada evento subsequente, que ocorre à medida que a requisição atravessa o sistema, até que a operação seja concluída (ou interrompida por algum erro). Cada ponto da jornada do *trace* é considerado um *span*, então, quando a requisição atravessa o

sistema, múltiplos *spans* são produzidos e cada um deles possui seu próprio identificador `span-id` e uma referência ao *span* anterior, que originou a chamada, `parent-id`. Este é basicamente o *span pai*, utilizado para conectar *spans* com seus respectivos antecedentes na cadeia de eventos.

O seguinte diagrama ilustra o funcionamento de *traces*:

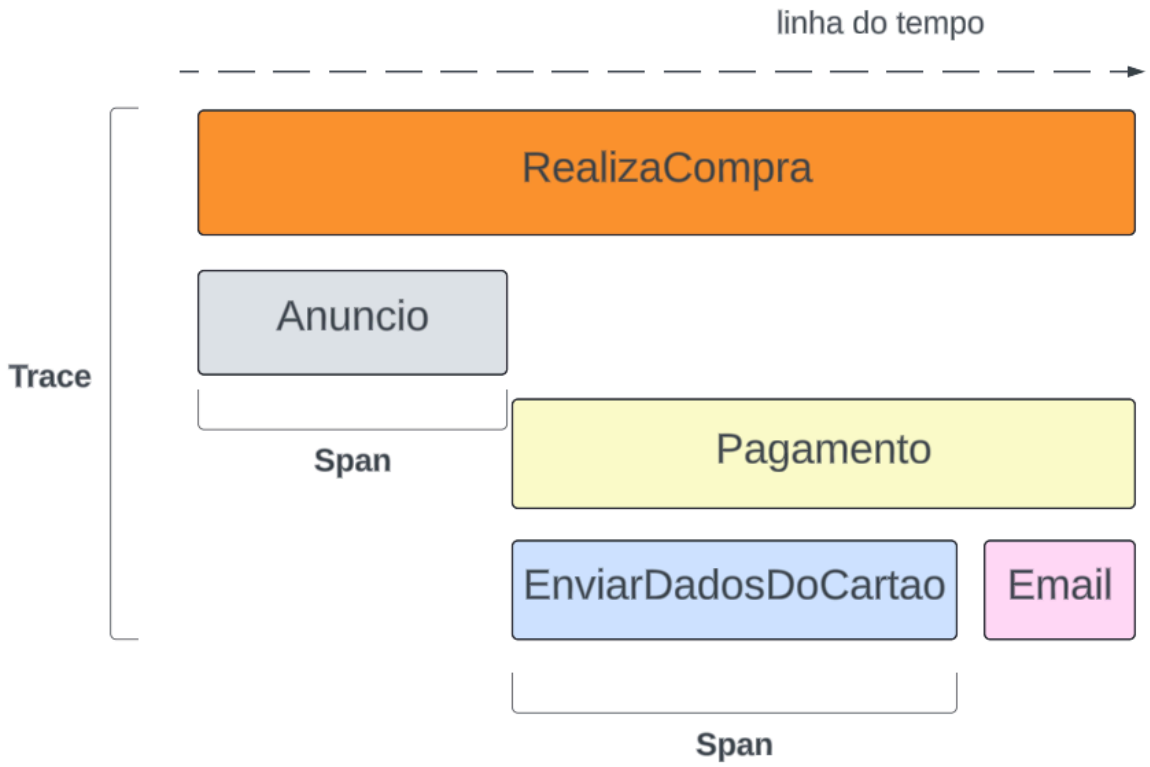

Figura 8.1: Ilustrando traces e spans

Essas informações são armazenadas em cabeçalhos HTTP (*HTTP Headers*), seguindo as recomendações do Consórcio World Wide Web (W3C). Mais detalhes podem ser encontrados na documentação oficial: https://www.w3.org/TR/trace-context/.

Os *spans* são emitidos quando alguma função é executada com o contexto de uma determinada requisição, portanto, eles possuem informações relacionadas àquela requisição específica, como: o caminho da URL, o verbo HTTP, o código de resposta HTTP. Os

autores(as) do código podem adicionar mais informações para enriquecer o rastreamento. Isso pode ser feito com *tags* ou pares de chave-valor que são aplicados como atributos adicionais (processo conhecido como *baggage*, cuja especificação pode ser encontrada aqui: https://w3c.github.io/baggage/).

Os *traces* e *spans* são emitidos por alguma função, servidor, VM, microsserviço, lambda ou qualquer outra entidade do sistema que realiza algum trabalho computacional, e são armazenados e gerenciados por ferramentas de rastreamento (como Jaeger, Tempo, Zipkin). Quando operadores(as) desejam consultar o rastreamento de algum serviço, consultas são processadas para obter os *traces* e seus respectivos *spans*, que são reunidos de acordo com seus identificadores e é assim que esses dados são organizados para fornecer este fantástico mecanismo de monitoramento.

8.1 Traces em ação

Temos que modificar a nossa aplicação de exemplo mais uma vez para produzir alguns *traces*. Vamos enviar rastros para o *OTel Collector*, que deve exportá-los para uma ferramenta de rastreamento Open Source chamada *Tempo* (https://grafana.com/oss/tempo/).

Mas, antes de explorarmos o código da aplicação, vamos alterar a configuração do *Collector* para definir um receptor, um exportador e uma pipeline para *traces*. O receptor (*receiver*) vai operar no protocolo OTLP (*OpenTelemetry Protocol*) portanto podemos defini-lo como `otlp/traces` com uma conexão gRPC cujo *endpoint* será `0.0.0.0:4417`. Logo depois, vamos definir um novo exportador (*exporter*) com o mesmo identificador `otlp/traces`, e o destino dos rastros será o *endpoint* `tempo:4517`, apontando para o contêiner `tempo` na porta 4517 (que será definido a seguir). Para concluir, adicionaremos uma terceira pipeline `traces` ao arquivo de

configuração, referenciando os devidos *receivers*, *processors* e *exporters*.

Aqui está o código do arquivo `collector-config.yaml` , para sua referência:

```
receivers:
  otlp:
    protocols:
      grpc:
      http:
  otlp/traces:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4417
  filelog:
    include: [ /var/log/sre/minhaApp.log ]

exporters:
  prometheus:
    endpoint: "0.0.0.0:9464"
  loki:
    endpoint: http://loki:3100/loki/api/v1/push
    default_labels_enabled:
      exporter: true
      job: true
  otlp/traces:
    endpoint: tempo:4517
    tls:
      insecure: true

processors:
  batch:
  resource:
    attributes:
    - key: service_name
      value: "metal-app"
      action: upsert
    - key: deployment_environment
```

```
      value: "dev"
      action: upsert
    - action: insert
      key: loki.resource.labels
      value: service_name, deployment_environment

service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [prometheus]
      processors: [batch]
    logs:
      receivers: [filelog]
      processors: [resource]
      exporters: [loki]
    traces:
      receivers: [otlp/traces]
      processors: [resource]
      exporters: [otlp/traces]
```

Com esta nova configuração definida, vá em frente e reinicie o contêiner do *Collector*:

```
docker stop collector

docker run --rm -d --name=collector -p 4317:4317 -p 4417:4417 -p 9464:9464
-p 55680:55680 -p 55681:55681 -v $(PWD)/collector-
config.yaml:/conf/collector-config.yaml -v $(PWD)/emissor-de-metricas-
simples/minhaApp.log:/var/log/sre/minhaApp.log --network sre-test-network
otel/opentelemetry-collector-contrib:latest --config /conf/collector-
config.yaml
```

Repare que o comando `docker run` contém o mapeamento de uma nova porta `-p 4417:4417` . Essa é a porta que foi definida na configuração do `exporter` inserida no passo anterior.

Para iniciar a solução de rastreamento *Tempo*, primeiro crie um arquivo de configuração `tempo.yaml` para especificarmos os parâmetros de operação desta ferramenta.

```
distributor:
  receivers:
      otlp:
        protocols:
          grpc:
            endpoint: 0.0.0.0:4517
storage:
  trace:
    backend: local
    local:
      path: /var/tempo/blocks

server:
    http_listen_port: 9085
```

Para mais informações sobre os parâmetros de configuração do *Tempo*, visite:

https://grafana.com/docs/tempo/latest/configuration/

Agora utilize o comando Docker a seguir para iniciar o contêiner *Tempo* com a configuração que criamos:

```
docker run --rm -d --name=tempo --mount source=tempo-
data,target=/var/tempo -v $(PWD)/tempo.yaml:/etc/tempo.yaml -p 4517:4517 -
p 9085:9085 --network sre-test-network grafana/tempo:latest -
config.file=/etc/tempo.yaml
```

E aqui está o código final da aplicação de exemplo. Lembre-se de executar o comando `go mod tidy` (para realizar o download dos módulos necessários para produzir *traces*) antes de proceder com `go run main.go .`

```
package main

import (
    "context"
    "fmt"
```

```
    "log"
    "net/http"
    "os"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"

    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    instrument "go.opentelemetry.io/otel/metric"
    otel_metric "go.opentelemetry.io/otel/metric"
    "go.opentelemetry.io/otel/sdk/metric"
    "go.opentelemetry.io/otel/sdk/resource"
    semconv "go.opentelemetry.io/otel/semconv/v1.17.0"

    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    "go.opentelemetry.io/otel/trace"
)

var meuContador otel_metric.Int64Counter

var tracer trace.Tracer

var (
    outfile, _ = os.Create("minhaApp.log")
    logger     = log.New(outfile, "", 0)
)

func main() {
    ctx := context.Background()

    serviceName := "metal-app"
    collectorAddress := "localhost:4317"
    logger.Printf("Establishing gRPC connection with %s...\n",
collectorAddress)
```

```
    dopts := []grpc.DialOption{
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithBlock(),
        grpc.WithTimeout(5 * time.Second),
    }
    conn, err := grpc.DialContext(ctx, collectorAddress, dopts...)
    if err != nil {
        log.Fatalf("ERROR: Unable to establish grpc connection to otel
metrics collector: %w", err)
    }

    exporter, err := otlpmetricgrpc.New(ctx,
otlpmetricgrpc.WithGRPCConn(conn))
    if err != nil {
        log.Fatalf("unable to create otel grpc exporter: %w", err)
    }

    var attrs []attribute.KeyValue
    // Always include service_name by default
    attrs = append(attrs, semconv.ServiceName(serviceName))

    res, err := resource.New(ctx, resource.WithAttributes(
        attrs...,
    ))
    if err != nil {
        log.Fatalf("failed to create resource: %w", err)
    }

    // metrics
    provider := metric.NewMeterProvider(
        metric.WithResource(res),
        metric.WithReader(metric.NewPeriodicReader(exporter)),
    )
    meter := provider.Meter("sre")

    c, err := meter.Int64Counter("sre.meu.contador")
    if err != nil {
        log.Fatalf("unable to create counter metric", err)
```

```
    }
    meuContador = c

    // traces
    collectorTracesAddress := "localhost:4417"
    t, _ := otlptracegrpc.New(ctx,
otlptracegrpc.WithEndpoint(collectorTracesAddress),
otlptracegrpc.WithInsecure())
    bsp := sdktrace.NewBatchSpanProcessor(t)
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithResource(res),
        sdktrace.WithSpanProcessor(bsp),
    )
    otel.SetTracerProvider(tp)
    tracer = otel.Tracer(serviceName)

    mux := http.NewServeMux()
    mux.Handle("/", otelhttp.NewHandler(otelhttp.WithRouteTag("/",
http.HandlerFunc(HelloServer)), "root", otelhttp.WithPublicEndpoint()))
    http.HandleFunc("/", HelloServer)
    http.ListenAndServe(":8080", mux)
}

func HelloServer(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    var attrs = []attribute.KeyValue{
        attribute.Key("operacao").String("oi"),
    }
    opt := instrument.WithAttributes(attrs...)

    logger.Print("emitindo metrica OTel...")
    meuContador.Add(ctx, 1, opt)

    fmt.Fprintf(w, OtherFunction(ctx, r.URL.Path[1:]))
}

func OtherFunction(ctx context.Context, nome string) string {
```

```
    _, span := tracer.Start(
        ctx,
        "digaOla",
        trace.WithAttributes(attribute.String("AlgumAtributo",
"QualquerValor")),
    )
     if span != nil {
        defer span.End()
    }

    return fmt.Sprintf("Olá, %s!", nome)
}
```

A aplicação precisa de novas dependências, portanto devemos importar os seguintes pacotes: `"go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"` para estabelecer a conexão gRPC com o *Collector*, `"go.opentelemetry.io/otel/sdk/trace"` para iniciar um processador de *traces* e `"go.opentelemetry.io/otel/trace"` para criar um processador e um provedor de *traces*. Para facilitar o acesso à instância de um rastreador nesta aplicação de exemplo, declaramos uma variável global `var tracer trace.Tracer` , que será referenciada entre funções diferentes.

Na função `main()` , antes de iniciarmos o servidor HTTP, temos um novo trecho de código (logo abaixo do comentário: `//traces`): uma variável é declarada para definir o endereço do *OTel Collector* (`collectorTracesAddress := "localhost:4417"`). Ela é utilizada como um argumento da função que inicia uma conexão gRPC (`otlptracegrpc.New`) e esta conexão é utilizada para criar um processador de *traces* (`bsp := sdktrace.NewBatchSpanProcessor(t)` . A função `sdktrace.NewTracerProvider()` recebe 3 parâmetros em forma de funções:

- `sdktrace.WithSampler(sdktrace.AlwaysSample())` : para definir as configurações de amostragem de rastreamento. A opção *AlwaysSample* faz com que 100% dos rastros sejam processados.

- `sdktrace.WithResource(res)` : fornece a instância de recurso (`res`) que contém a *label* `service_name` (*convenção semântica*), que também é utilizada no `NewMeterProvider()` , como explicado no capítulo 6.
- `sdktrace.WithSpanProcessor(bsp)` : adiciona o processador de *traces* ao provedor.

Nas próximas linhas de código, definimos o provedor de *traces* global com `otel.SetTracerProvider(tp)` e populamos nossa variável `tracer` com o rastreador obtido através da função `otel.Tracer(serviceName)` . Com estas instruções executadas antes da inicialização do servidor HTTP, toda vez que um rastreamento ocorrer em qualquer função, os *traces* e seus respectivos *spans* serão emitidos para o *Collector* através do rastreador (`tracer`).

Existe uma pequena mudança na função `HelloServer()` (que é aquela que é executada em cada requisição enviada para `http://localhost:8080/sre`). Essa função agora invoca outra função `OtherFunction(ctx, r.URL.Path[1:])` , que recebe dois argumentos (uma instância de um contexto e o fragmento do caminho da URL) e retorna a mensagem que deve ser apresentada para o usuário. Aqui está o código da nova função:

```
func OtherFunction(ctx context.Context, nome string) string {
    _, span := tracer.Start(
        ctx,
        "digaOla",
        trace.WithAttributes(attribute.String("AlgumAtributo",
"QualquerValor")),
    )
     if span != nil {
        defer span.End()
    }

    return fmt.Sprintf("Olá, %s!", nome)
}
```

A função recebe um contexto (`ctx context.Context`), cuja instância foi definida na função anterior que a invocou (`HelloServer()`) e referencia o contexto da requisição (`ctx := r.Context()`). Cada requisição tem seu próprio contexto individual, que contém metadados que serão associados com os rastros. Note que a primeira coisa que a função faz é iniciar o rastreamento invocando `tracer.Start()` com 3 argumentos: o contexto da requisição `ctx` , uma string `digaOla` (que será o nome do *span* gerado pela execução desta função) e uma lista de atributos contendo apenas um único par de chave e valor `trace.WithAttributes(attribute.String("AlgumAtributo", "QualquerValor"))` . Em aplicações reais estes atributos podem enriquecer ainda mais as capacidades de observabilidade de seu sistema com informações relacionadas ao usuário, às operações ou até mesmo dados de infraestrutura (você pode, por exemplo, inserir um atributo relacionado ao tipo de servidor que está processando a requisição e correlacionar falhas ou problemas de performance ao observar rastros).

A função `tracer.Start()` retorna um contexto e um *span* (`context.Context, trace.Span`). O código ignora o contexto que é retornado (`_, span`), já que nesta simples aplicação de exemplo não vamos executar mais do que duas funções para cada requisição (e nem vamos enviar uma nova requisição a uma segunda aplicação).

> A variável que corresponde ao contexto retornado pela função `Start()` é declarada com um *underscore* (_) para evitar um erro de compilação. A linguagem de programação Go não permite que variáveis sejam definidas sem serem usadas, portanto esta é a sintaxe necessária para ignorar a variável que não será usada.

As próximas linhas de código vão apenas verificar se o *span* não é nulo, registrar o fim do *span* (`span.End()`) e retornar o texto que diz `"Olá, <fragmento_da_URL>!"` . Note que o fim do *span* só será registrado quando a função `OtherFunction()` for concluída, e isso é feito através

de um *adiamento* realizado por uma funcionalidade da linguagem Go. Quando um `defer` é declarado antes da chamada de uma função, ela só é executada quando a função na qual o `defer` se encontra completa sua execução.

Nosso próximo passo é a criação de um novo *data source* no Grafana. O procedimento é bem similar ao que foi descrito nos capítulos anteriores: cheque se o contêiner `grafana` está em execução, navegue para `http://localhost:3000/connections/datasources`, clique em *Add new data source*, selecione a opção *Tempo* e, quando a tela de criação de *data source* surgir, preencha o campo *URL* com o endereço `http://tempo:9085`. Após rolar a página até o final, clique em *Save & test* para concluir este passo.

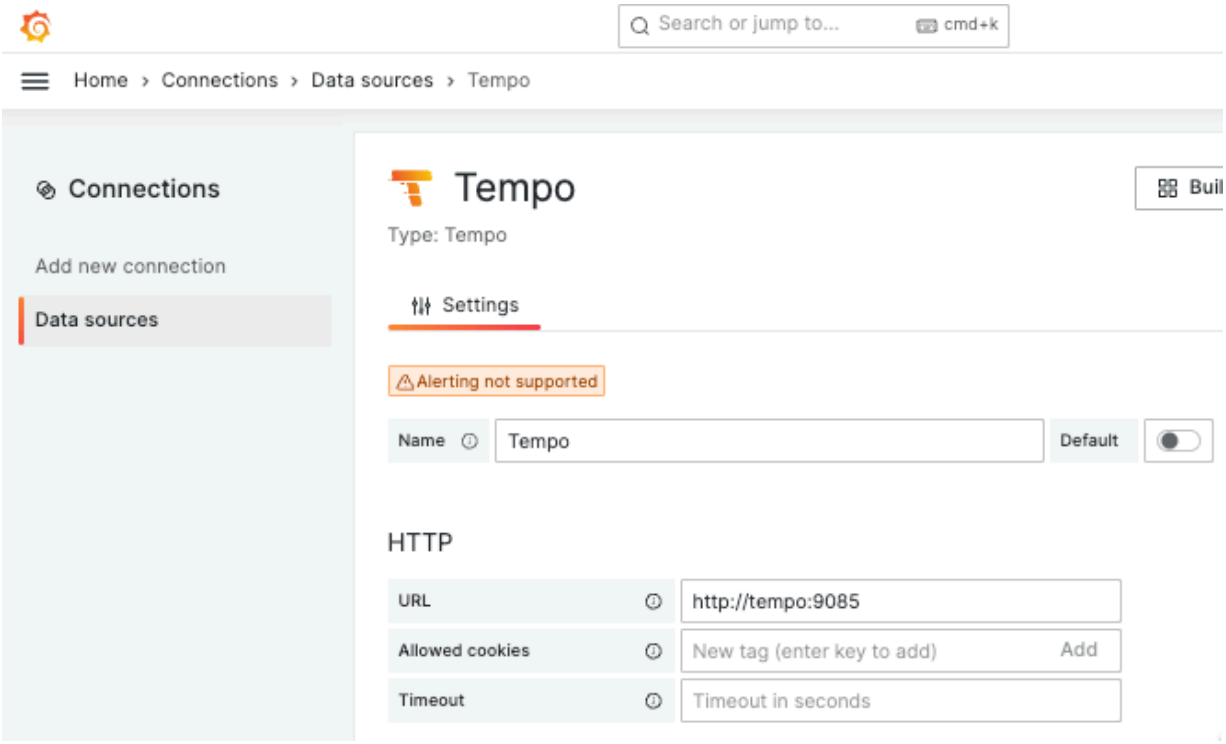

Figura 8.2: Criando um data source do tipo Tempo em Grafana

Agora vamos ver os *traces* em ação. Navegue para `http://localhost:3000/explore` e selecione *Tempo* no campo de seleção de *data sources* no topo da página do lado esquerdo. Essa ação exibe os campos de consulta do *Tempo* logo abaixo com a aba

TraceQL selecionada por padrão. A linguagem de consulta TraceQL (*Tempo Query Language*) é utilizada para consultar *traces* e possui uma sintaxe similar ao PromQL, por exemplo: `{.service_name="metal-app"}` deve retornar os *traces* produzidos por nossa aplicação de exemplo (mais detalhes sobre a sintaxe podem ser encontrados em https://grafana.com/docs/tempo/latest/traceql/).

Para simplificar nosso experimento, clique na aba *Search* e observe os campos que podem ser usados para filtrar *traces*. Veja que você pode selecionar nossa aplicação no campo *Service Name* e o *span* produzido pela função `OtherFunction()` pode ser encontrado no campo *Span Name*. Execute uma consulta e clique em qualquer *trace* para ver o resultado.

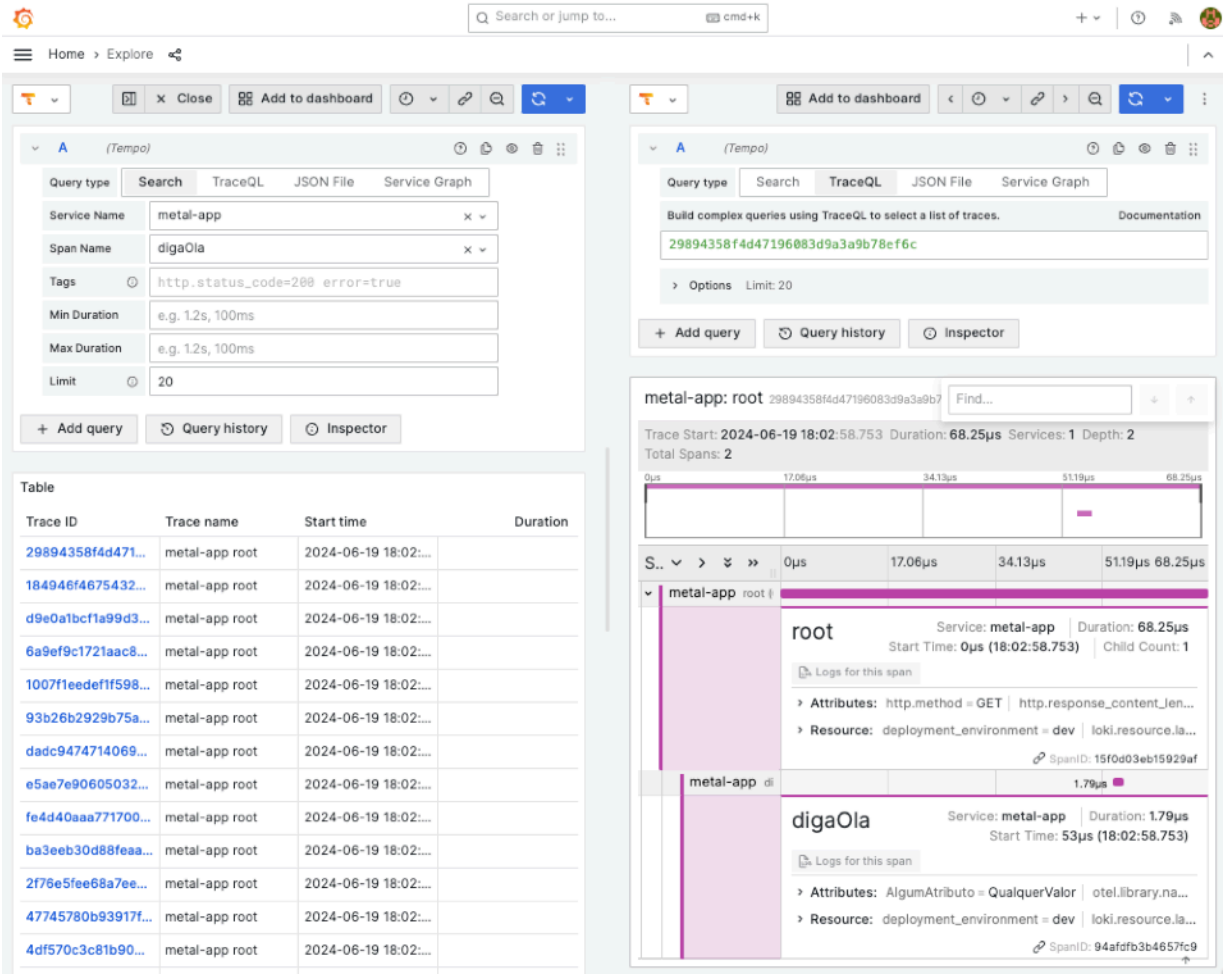

Figura 8.3: Analisando resultados de uma consulta ao Tempo

Veja que no lado direito da tela existem dois *spans*: um para a função de roteamento da requisição HTTP `HelloServer()` que está associada com o caminho raiz da URL `/` (*root*); e a outra função `OtherFunction()`, onde criamos um *span* explicitamente e adicionamos um atributo customizado. Note que os *spans* possuem metadados relacionados à requisição, como `http.target = "/sre"`, `http.method = "GET"` ou `http.status_code = 200`, e cada função que é rastreada pode adicionar mais atributos customizados para facilitar a análise de alguma jornada de usuário. Esta alta cardinalidade é esperada porque nem todos os *traces* precisam ser registrados, já que é comum que se configure uma *amostragem*.

8.2 Amostragem de Traces

Para facilitar o gerenciamento de *traces* sem exigir muita infraestrutura para armazenar uma quantidade massiva de dados e tornar consultas mais eficientes, a técnica de *amostragem* (em inglês, *sampling*) é utilizada para manter apenas os *traces* que são mais relevantes para fins de manutenção e suporte do serviço.

Se uma aplicação processa centenas de milhares de requisições a cada segundo, não é muito útil registrar cada rastro produzido por cada uma dessas requisições (e também não é muito eficiente em termos de custos de infraestrutura), portanto a biblioteca de código fornecida pelo OpenTelemetry permite que os operadores(as) implementem tal *amostragem* de acordo com suas preferências.

Vamos ilustrar esse conceito criando o nosso próprio *sampler* em nossa aplicação de exemplo. Abra novamente o código do nosso script `main.go` e adicione o seguinte código logo depois da seção de `import`:

```
type IgnoreCaminhoSampler struct{}

func (ics *IgnoreCaminhoSampler) ShouldSample(p
```

```
sdktrace.SamplingParameters) sdktrace.SamplingResult {
    result := sdktrace.SamplingResult{
        Tracestate:
trace.SpanContextFromContext(p.ParentContext).TraceState(),
    }
    shouldSample := true
    for _, att := range p.Attributes {
        if att.Key == "http.target" && att.Value.AsString() == "/meh" {
            shouldSample = false
            break
        }
    }
    if shouldSample {
        result.Decision = sdktrace.RecordAndSample
    } else {
        result.Decision = sdktrace.Drop
    }
    return result
}

func (ics IgnoreCaminhoSampler) Description() string {
    return "excludeBasedOnURLPath"
}
```

Aqui nós estamos criando um *struct* que implementa a interface `sdktrace.Sampler` (https://pkg.go.dev/go.opentelemetry.io/otel/sdk/trace#Sampler). Em Go, para implementar uma interface, você precisa fazer com que um *struct* implemente suas funções, por isso, o nosso `IgnoreCaminhoSampler` tem duas funções: `ShouldSample()` e `Description()`.

A primeira função é onde a mágica acontece, é aqui que devemos decidir se o *trace* será incluído na amostragem ou não. Neste exemplo nós inspecionamos os metadados da requisição iterando sob todos os atributos (`for _, att := range p.Attributes`) e determinamos se o *trace* deve ser registrado na amostragem (`sdktrace.RecordAndSample`) ou se deve ser ignorado (`sdktrace.Drop`). A

condição implementada aqui é baseada no cenário em que a chave do atributo é `http.target` e o valor correspondente a este atributo é `"/meh"`, fazendo com que as requisições HTTP cujo alvo seja `/meh` tenham seus *traces* ignorados, ou seja, o nosso *OTel Collector* não vai exportar esses *traces* para o *Tempo*.

Esse novo *sampler* deve ser adicionado como um parâmetro do construtor `sdktrace.NewTracerProvider()` que se encontra dentro da função `main()`. Aqui está o código com essa nova configuração:

```
bsp := sdktrace.NewBatchSpanProcessor(t)
ignoreCaminhoSampler := new(IgnoreCaminhoSampler)
tp := sdktrace.NewTracerProvider(
    sdktrace.WithSampler(ignoreCaminhoSampler),
    sdktrace.WithResource(res),
    sdktrace.WithSpanProcessor(bsp),
)
otel.SetTracerProvider(tp)
tracer = otel.Tracer(serviceName)
```

Reinicie a aplicação de exemplo e execute os seguintes comandos para produzir algumas requisições HTTP:

```
for i in {1..10}; do curl -s "http://localhost:8080/sre"; done
```

E, logo depois, repita o comando, mas desta vez modifique a URL para enviar requisições para o caminho `/meh`:

```
for i in {1..10}; do curl -s "http://localhost:8080/meh"; done
```

Navegue para o console do Grafana novamente: `http://localhost:3000/explore`. No formulário de busca de *traces*, adicione `http.target="/sre"` no campo *Tags* para filtrar resultados e veja que alguns *traces* aparecem. Depois, modifique o filtro, tente encontrar *traces* com a tag `http.target="/meh"` e veja que nenhum *trace* será retornado na busca.

Agora você deve estar se perguntando: "Por que não mudamos a função do nosso *sampler* para rastrear apenas as requisições com

erros?". Isso é uma ideia interessante, já que SREs devem focar em eventos que impactam os clientes de um determinado serviço, logo, uma falha no sistema é bastante relevante e deve ser incluída na amostragem. Porém, isso não é possível dentro da função `ShouldSample()`, pois ela está agindo em cada *span* à medida que cada requisição atravessa o sistema, o que ocorre antes que a *resposta* HTTP seja produzida. Em outras palavras, o *sampler* implementado no código da aplicação age na *entrada* da requisição e não na *saída*.

Quando realizamos a amostragem de *traces* depois que cada transação processa a cadeia de *spans* até o final da jornada de uma requisição, diz-se que esta é uma *amostragem de cauda* (*tail sampling*), pois opera no *final* do rastreamento. Para implementar um *tail sampling*, devemos adicionar mais parâmetros para o arquivo de configuração do nosso *OTel Collector*. Aqui está um exemplo que você pode adicionar no arquivo `collector-config.yaml`:

```
processors:
  # ...
  tail_sampling:
    decision_wait: 10s
    policies:
      [
        {
          name: errors-policy,
          type: status_code,
          status_code: { status_codes: [ERROR] }
        },
        {
          name: randomized-policy,
          type: probabilistic,
          probabilistic: { sampling_percentage: 25 },
        },
      ]
```

Nesse novo *processor*, o primeiro parâmetro configura o *OTel Collector* para que se espere 10 segundos para decidir se o *trace*

deve ser incluído na amostragem logo depois que o primeiro *span* é gerado (`decision_wait: 10s`). O próximo bloco define as políticas (`policies`) de amostragem do *tail sampling*:

- O primeiro critério é baseado em códigos de status (`status_code`) que contêm erros (`[ERROR]`).
- A segunda política é do tipo "probabilístico" (`probabilistic`) e vai adicionar quaisquer *traces*, de forma randômica, registrando apenas 25% de todos os *traces* (`sampling_percentage: 25`).

Dessa forma, os operadores(as) podem inspecionar *todos os erros* (100%) e apenas um percentual menor de todos os outros *traces* (25%), o que pode ser útil para analisar outros casos que não envolvam erros, como problemas de performance (tempo de resposta), por exemplo. Para mais informações sobre o `tail_sampling` *processor* e cada um de seus parâmetros, acesse https://github.com/open-telemetry/opentelemetry-collector-contrib/blob/main/processor/tailsamplingprocessor/README.md

Esta configuração deve ser adicionada no bloco de `processors` junto às outras configurações que foram definidas nos capítulos anteriores (`batch` e `resource`):

```
processors:
  batch: ...
  resource: ...
  tail_sampling: ...
```

E este novo *processor* deve ser declarado na pipeline de `traces` da seguinte forma: `processors: [resource,tail_sampling]` . Aqui está o bloco de configuração do serviço *OTel Collector* com suas respectivas pipelines:

```
service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [prometheus]
```

```
      processors: [batch]
    logs:
      receivers: [filelog]
      processors: [resource]
      exporters: [loki]
    traces:
      receivers: [otlp/traces]
      processors: [resource,tail_sampling]
      exporters: [otlp/traces]
```

Uma vez que o arquivo `collector-config.yaml` for atualizado com estas novas instruções, você pode testar as políticas de amostragem fazendo com que sua aplicação produza erros de propósito. Aqui está um trecho de código que pode ser inserido na função `HelloServer()` para produzir uma resposta HTTP 500 (*internal server error*). Esta lógica verifica se o caminho da URL é `/erro` e retorna um erro ao usuário:

```
if r.URL.Path[1:] == "erro" {
    w.WriteHeader(http.StatusInternalServerError)
     _, _ = w.Write([]byte("deu errado"))
    return
```

Reinicie o *OTel Collector* para que possa utilizar a nova configuração, execute a aplicação de exemplo com este novo código e envie algumas requisições para testar a proporção de *traces* para cada um dos casos:

```
% for i in {1..100}; do curl -s "http://localhost:8080/sre"; done
% for i in {1..100}; do curl -s "http://localhost:8080/erro"; done
% for i in {1..100}; do curl -s "http://localhost:8080/meh"; done
```

Com este conhecimento, você pode utilizar métricas, *logs* e *traces* para o monitoramento dos serviços da sua organização, discutir estratégias para Observabilidade e ajudar desenvolvedores com o código necessário para obter tal telemetria.

No próximo capítulo vamos conhecer os conceitos mais importantes para a prática de SRE, que vão ajudar a colocar este conhecimento

de observabilidade em prática para medir a confiabilidade de serviços e como discutir esses conceitos com outros membros da organização.

Parte 4: Conceitos de confiabilidade

Capítulo 9
Objetivos de Nível de Serviço (SLOs)

Não se deve esperar que sistemas funcionem perfeitamente o tempo todo. Esta expectativa não é realista para serviços, especialmente para aqueles que oferecem funcionalidades computacionais que são executadas em sistemas operacionais com múltiplas bibliotecas de código, operando em servidores ou microcontroladores cujas placas, circuitos e periféricos estão em constante atividade, conectados através de redes com cabos físicos, ondas de rádio, constituindo um universo de dependências físicas e virtuais que está sujeito a eventuais falhas.

Ao trabalhar com tais sistemas, devemos internalizar o fato de que usuários(as) possuem uma tolerância natural para comportamentos irregulares que ocorrerem em pequenas iterações ou em um curto espaço de tempo. Eventos inesperados que ocorrem esporadicamente podem ser considerados insignificantes em relação a uma grande carga de trabalho ou quando um sistema processa um número substancial de transações.

Ao encontrar um erro, ou mesmo se alguma operação demorar mais do que o normal para produzir uma resposta, usuários(as) não vão abandonar o website ou aplicativo móvel imediatamente. Isso ocorre apenas quando clientes acumulam uma frustração relacionada à frequência de tais erros ou problemas de performance, portanto, tais eventos indesejados têm que ser percebidos recorrentemente por um espaço de tempo significativo para que se resulte em desistência ou vontade de procurar competidores que oferecem serviços semelhantes.

A ideia de que algo deve funcionar 100% do tempo não está de acordo com as práticas de SRE, a Engenharia de Confiabilidade possui diretrizes para estabelecer expectativas perante os clientes e orientar as prioridades dos provedores do serviço. Essas diretrizes definem uma frequência em que o serviço deve operar dentro de certas condições alinhadas com a perspectiva do cliente, tal frequência é geralmente definida com um número de "noves", por exemplo: 99.9% ou 99.999%, representando o tempo em que o serviço opera de forma esperada.

Existem dois métodos para expressar quais devem ser as expectativas de performance de um serviço:

- O Acordo de Nível de Serviço (*Service Level Agreement* ou SLA)
- O Objetivo de Nível de Serviço (*Service Level Objective* ou SLO)

O SLA é o acordo propagado externamente, aquele que é divulgado para clientes e que pode resultar em penalidades financeiras para o provedor do serviço caso os critérios de performance não sejam compridos. Um SLA é declarado em contratos para divulgar um comprometimento com um certo nível de performance, seja com uma descrição do que esperar em termos de disponibilidade (*uptime*) ou tempo de resposta (*latency*), criando uma relação de confiança com o cliente e definindo expectativas claras sobre seu serviço.

Como discutido brevemente no capítulo 5, ao descrever uma das soluções da nuvem (AWS EC2), se a organização não cumprir o SLA de *uptime*, por exemplo, o cliente pode receber descontos ou um crédito para utilizar mais recursos ou funcionalidades oferecidas pelo provedor do serviço, ou seja, a organização enfrenta um impacto financeiro se não cumprir o acordo. Este impacto, por sua vez, gera um incentivo para mais investimento em práticas de confiabilidade para evitar incidentes recorrentes que possam frustrar clientes.

Neste livro não vamos entrar em detalhes burocráticos de SLA, pois nem toda organização ou time de engenharia de software precisa

adotar SLAs e formalizar contratos para disponibilizar seus serviços para clientes, sejam eles internos ou externos. O nosso foco será no Objetivo de Nível de Serviço (SLO), pois pode ser implementado por qualquer time de engenharia sem atritos contratuais.

9.1 SLO

O SLO é o método utilizado para definir um objetivo internamente, que não é divulgado para clientes em contratos e não tem consequências financeiras arbitrariamente definidas. O SLO é um acordo que serve como orientação para que o time responsável por um determinado serviço tome decisões sobre as tarefas que devem ser priorizadas ou fique ciente de oportunidades para tentar mudanças potencialmente disruptivas. Estes métodos não são mutualmente exclusivos, ou seja, uma organização pode, por exemplo, estabelecer um SLA de 99.9 % e um SLO de 99.95 % para que o acordo interno seja um pouco mais restrito que o externo, fazendo com que os times de engenharia tomem ações para melhorar a confiabilidade antes que futuros eventos possam violar o SLA.

Para deixar claro: ambos, SLA e SLO, definem a frequência em que um serviço deve operar de uma maneira satisfatória aos seus clientes. Porém, SLA é um acordo externo formalizado com entidades fora da organização, enquanto o SLO é um objetivo interno que auxilia a organização na tomada de decisões de engenharia.

Como se define um SLO?

O Objetivo de Nível de Serviço está relacionado à expectativa de encontrar um certo número de eventos bons em relação ao número total de eventos em um espaço de tempo. Para ilustrar matematicamente: `(eventos bons ÷ número total de eventos) x 100` .

Então, se um serviço processa 3000 transações e produz 3 erros, de acordo com os *datapoints* de métricas emitidas dentro de um minuto, encontramos o seguinte resultado: `(2997 / 3000) * 100 = 99.9` , o que significa que, em um cenário onde o SLO é 99.9 %, este serviço não violou o acordo naquele minuto de operação.

O espaço de tempo em que um SLO é observado nos ajuda a expressar a frequência em que o serviço deve operar de forma confiável. Um acordo de 99%, por exemplo, acomoda eventos ruins por 14 minutos e 24 segundos *por dia*, pois:

- 1 dia tem 1440 minutos
- 1% de 1440 é igual a 14.4, o que corresponde a 14 minutos e uma fração de 0.4
- 0.4 minutos convertidos em segundos (0.4 * 60) resulta em 24 segundos

Isso quer dizer que este serviço possui uma cota diária de `14m e 24s` em que ele pode produzir erros, apresentar lentidão ou ficar fora do ar (*downtime*), isso pode ser totalmente aceitável dependendo do serviço e de como ele é utilizado por clientes.

Esse exemplo descreve uma cota diária, mas SLOs podem ser definidos com diferentes períodos de observação. Se considerarmos uma janela semanal, o mesmo acordo de 99% se traduz em `1h, 40m e 48s` de tempo permitido durante a semana em que se acomodam eventuais falhas ou procedimentos de manutenção. Os eventos de manutenção planejados devem fazer uso destes acordos, por exemplo, uma atualização de servidores, algum ajuste nas configurações de infraestrutura, algum *script* que deve ser executado em um banco de dados que deixe o sistema parcialmente indisponível etc.

Essa janela de tempo do SLO depende da estratégia de SRE aplicada para o serviço e como suas funcionalidades e jornadas de usuários(as) estão alinhadas com a organização. Uma janela de

avaliação semanal é geralmente uma boa escolha, mas para serviços com transações esporádicas, que não são utilizados com frequência ou que produzem poucos *datapoints* em suas métricas, pode-se utilizar uma janela mensal ou até mesmo observar o SLO em espaços de 3, 6 ou 12 meses.

A tabela a seguir mostra alguns SLOs comuns e seus respectivos períodos de indisponibilidade permitidos:

Confiabilidade	**99%**	**99.9%**	**99.99%**
dia	14m 24s	1m 26.4s	8.6s
semana	1hr 40m 48s	10m 4.8s	1m 0.5s
mês	7hr 18m 17.5s	43m 49.7s	4m 23s
ano	3d 15hr 39m 29s	8h 45m 57s	52m 35.7s

Como você pode ver, o número de noves é diretamente proporcional à confiabilidade esperada por um serviço, porém, não devemos definir uma quantidade de noves muito alta para o SLO de um serviço sem entender suas condições. Além de reunir detalhes sobre o serviço, suas funcionalidades e seu impacto organizacional, deve-se levar em consideração a resiliência de suas dependências, as capacidades e os custos de sua infraestrutura, bem como as consequências que um SLO "apertado" pode acarretar para o time que ficará de plantão (*on-call*) e será notificado por alarmes relacionados a qualquer eventual problema neste serviço (vamos discutir esta questão de alarmes com mais detalhes no próximo capítulo.). Um serviço com muitos noves exige uma arquitetura resiliente, escalável, segura, com um custo maior de infraestrutura e processos amadurecidos de automação e resolução de problemas.

SLOs devem ser definidos e ajustados colaborativamente. SREs devem trabalhar com outros membros da organização para mapear

os serviços que compreendem as funcionalidades críticas de um sistema, encontrando as peças fundamentais do produto principal de uma organização e, se necessário, agendar reuniões com os responsáveis por cada uma das funcionalidades para aprender mais sobre os serviços aos quais proverão suporte. Tais discussões sobre SLO envolvem uma análise de quão críticas são as funcionalidades do serviço, sempre tentando incluir os especialistas no assunto, ou, do inglês: *Subject Matter Experts*, ou apenas SMEs, que combinam o seu entendimento profundo do produto, suas funcionalidades e qualquer consequência financeira em potencial relacionada ao impacto aos usuários (por exemplo: qual seria o impacto financeiro estimado se o sistema de processamento de pedidos ficar fora do ar por 2 horas?).

Ao navegar uma implementação de SRE em seus estágios iniciais, para evitar muito atrito e "paralisia analítica" ao tentar definir um SLO inicial para um serviço, SREs podem escolher um ponto de partida, como três noves (99.9%), por exemplo, e acompanhar a performance dos serviços para que ele possa ser ajustado depois. Os ajustes de SLO são efetuados ao observar como o serviço se comporta durante eventos disruptivos como *bugs* introduzidos durante ciclos de *release*, problemas de escalabilidade em decorrência de campanhas de marketing (disparos de notificações), falhas em terceiros, problemas de infraestrutura e quaisquer outras dependências.

Uma maneira de refletir sobre o número ideal para o SLO é considerar o tempo médio entre falhas, cuja sigla, em inglês, é MTBF (*Mean Time Between Failures*). O MTBF é baseado nos períodos em que o serviço opera de maneira estável entre incidentes, é uma média dos espaços de tempo em que o serviço não apresenta incidentes. Portanto, um serviço com um MTBF alto tem uma boa confiabilidade e pode ter seu SLO ajustado de acordo com ele.

Um ponto bastante importante para ser lembrado é que os clientes não vão perceber a diferença entre utilizar um serviço que opera com 3 noves ou 4 noves, portanto SREs devem educar outros times

sobre como o investimento em aumentar o número de noves, na maioria dos casos, não é justificado, e os membros da organização devem encontrar um equilíbrio entre o que faz sentido em termos de manter os clientes felizes e os esforços operacionais e o custo aplicado do lado da engenharia.

9.2 SLIs

Como visto anteriormente, o SLO é a frequência em que um serviço opera de maneira satisfatória. O critério desta satisfação é composto por um ou mais Indicadores de Nível de Serviço (*Service Level Indicators* ou apenas SLIs). Os SLIs são as condições que determinam se o serviço está operando como esperado. Tais condições são definidas de acordo com as características do serviço que entregam valor ao cliente e podem ser observadas através de métricas, geralmente utilizando os 4 sinais de ouro de SRE como referência: latência, tráfego, erros e saturação.

Vamos trazer esses conceitos para algo mais próximo da realidade de serviços Web, descrevendo SLIs e o SLO de um website X. Vamos começar definindo dois SLIs baseados em condições de erros e tempo de resposta:

- SLI 1: As requisições direcionadas ao Website não devem retornar erros (Código de resposta HTTP != 5xx).
- SLI 2: As requisições direcionadas à página inicial do website devem ser processadas em até 400ms (tempo de resposta < 400ms).

Agora vamos supor que SREs discutiram um objetivo interno com os times responsáveis pelo Website X, incluindo gerências de produto, marketing, finanças etc., chegando à conclusão de que um SLO de 99.9%, observado em uma janela de tempo de 7 dias, é um bom

alvo para fins de suporte e manutenção da qualidade do website. Portanto, podemos expressar o seguinte acordo interno:

- 99.9% das requisições direcionadas ao Website não devem retornar erros (`Código de resposta HTTP != 5xx`).
- 99.9% das requisições direcionadas à página inicial do website devem ser processadas em até 400ms (tempo de resposta < 400ms).
- Tais condições devem ser observadas em uma janela de 7 dias.

Repare que os SLIs são condições para identificar se o serviço está operando como desejado e o SLO agrupa todos essas condições em um nível mais amplo relacionado à frequência com que as condições devem ser mantidas. Em outras palavras, podemos visualizar a hierarquia desses conceitos imaginando o SLO como um círculo que contém um ou mais SLIs.

É uma prática comum calcular SLOs em termos de minutos bons e minutos ruins, sendo que um minuto pode ser considerado ruim se possuir um número considerável de eventos ruins. Para serviços de alta escalabilidade, com milhões de usuários e centenas de milhares de transações por segundo, por exemplo, pode ser mais estratégico calcular com uma granularidade maior, potencialmente em janelas de 30 segundos ou menores do que 10 segundos, dependendo do serviço, da capacidade do banco de dados de série temporal e da infraestrutura do sistema de monitoramento. Vamos supor o seguinte cenário em que os alertas associados com o Website X dispararam duas vezes em um período de 7 dias e, após uma triagem do ocorrido, foram registradas as seguintes janelas de tempo em que o serviço acumulou minutos ruins:

- 00:10-00:26: Alto número de *errors* produzidos terça-feira de madrugada (Duração do evento: 16 minutos)
- 22:10-22:07: Problemas de performance (alta latência) encontrada quarta-feira à noite (Duração do evento: 7 minutos)

Isso significa que dois SLIs diferentes foram violados em diferentes momentos nesta semana, se considerarmos o número total de minutos em uma semana (10080), então podemos verificar se o Website X quebrou o acordo de 99.9% definido em seu SLO calculando se o número de minutos ruins neste período de 7 dias excedeu o número permitido. Se calcularmos 99.9% de 10080, obtemos 10069.92, o que significa que o serviço Website X não pode apresentar uma confiabilidade semanal abaixo de 10069 minutos (estamos focando em eventos agregados por minuto, portanto vamos ignorar os valores decimais). Cada minuto ruim identificado pelas condições violadas de cada SLI é acumulado neste cálculo, portanto, temos um total de 23 minutos ruins. Se subtrairmos 23 de 10080, encontramos 10057, que está abaixo de 10069, ou seja, se aplicarmos a equação do SLO (`(eventos bons ÷ número total de eventos) x 100`), isso quer dizer que `( 10057 / 10080 ) * 100 = 99.77` , portanto este número de minutos ruins violou o SLO de 99.9 % desta semana.

> Se múltiplas condições forem violadas no mesmo minuto, estes eventos ruins serão contabilizados como um único minuto ruim. SLIs funcionam da mesma maneira que uma operação lógica "OU" (*OR*), onde se permite que quaisquer ou todas as partes de múltiplas condições sejam verdadeiras.

Esta seção explorou SLIs de um website X como um exemplo para ilustrar os conceitos que compreendem a implementação de SLOs, mas outros serviços baseados em tecnologias diferentes como *pipelines* de dados, sistemas de processamento de lote (*batch*) ou aplicações de *Machine Learning*, podem utilizar diferentes condições para a avaliação de performance sob a ótica de seus respectivos clientes e dependências de infraestrutura.

Aplicativos móveis podem utilizar um SLI baseados em número de interrupções (*crashes*), por exemplo, para implementar SLOs mais alinhados com seu comportamento em relação à suas respectivas

plataformas (Android e iOS) e a estabilidade da aplicação ao operar sob elas.

A seguir, vamos apresentar recomendações de SLIs. Mais especificamente, os SLIs mais comuns utilizados por serviços baseados em sistemas Web são:

- Taxa de Sucesso (*Success Rate*) e
- P99 da Latência (*P99 Latency*)

Taxa de Sucesso

O SLI *Success Rate* (ou apenas SR) é um indicador recomendado para determinar se um serviço está produzindo um número considerável de erros em um determinado período. Este SLI é baseado na relação entre o número de requisições bem-sucedidas e o total de requisições, que pode ser expressado através da seguinte equação: `SR (Success Rate) = 100 * (1 - (número de erros / número total de requisições ))`.

Talvez você já tenha encontrado algum alarme configurado para alertar membros da engenharia se o número de erros produzidos por um serviço ultrapassar algum valor fixo, algo como: `numero de http 5XX's > 100`, por exemplo. Essa abordagem não é recomendada pois esta condição está sujeita a ser revisitada com frequência à medida que novos padrões de tráfego do serviço são introduzidos e o limiar de erros oscilar. Isso resulta também em uma sobrecarga de comunicação quando o time *on-call* precisar discutir a sensibilidade do alarme e se é realmente necessário enviar um *page* para acordar alguém as 3 da manhã se o serviço produzir 100 erros enquanto processava 100.000 transações. Esta é a vantagem da Taxa de Sucesso, pois trabalha com números relativos e facilita a análise do impacto de alguma degradação momentânea.

A Taxa de Sucesso também fornece uma visualização bastante conveniente para SREs, especialmente em gráficos que mostram

quando o serviço viola acordos, a janela de um incidente, e a recuperação do serviço uma vez que um incidente é mitigado, ou se o serviço for restaurado automaticamente. Tais gráficos podem mostrar também as janelas de tempo utilizadas para o cálculo do Tempo Médio de Detecção (do inglês, MTTD ou *Mean Time To Detect*), que é o tempo médio que se leva para detectar algum problema com o serviço, e o Tempo Médio de Recuperação (MTTR ou *Mean Time To Recover*), que expressa o tempo médio para que algum membro da engenharia aplique uma mitigação ou para que o serviço volte a operar dentro da performance esperada através de alguma automação, restaurando a experiência do usuário. Veja no gráfico a seguir.

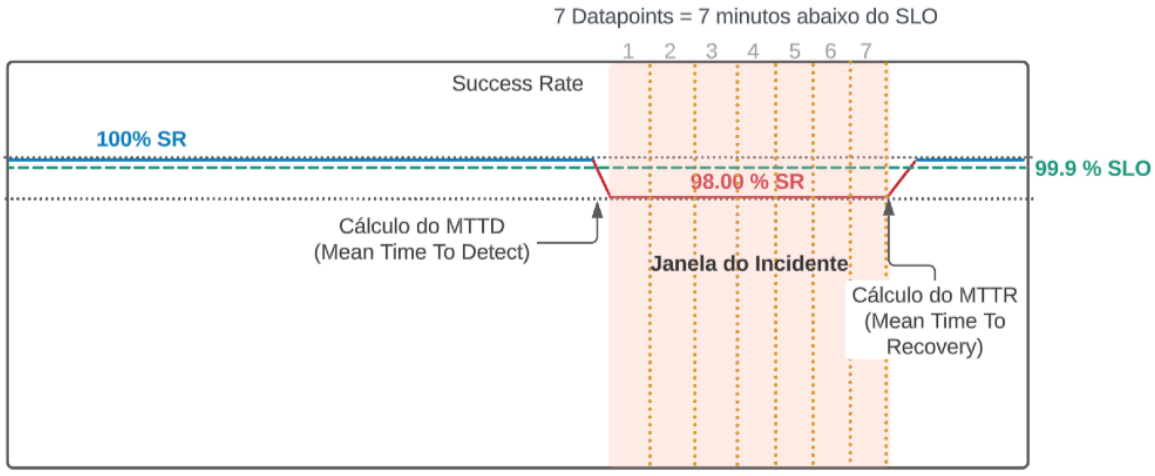

Figura 9.1: Taxa de Sucesso em gráfico em relação ao SLA ou SLO

Estas métricas de MTTD e MTTR são bem úteis para avaliar se a organização precisa dedicar mais recursos para acelerar o processo de detecção de problemas, para amadurecer processos para encontrar mitigações de forma mais rápida ou desenvolver automações de restauração de serviços para encurtar cada vez mais as janelas de incidentes subsequentes. O gráfico da Taxa de Sucesso é um ótimo recurso para analisar a duração de cada um destes períodos (deteção e restauração) e serve como um bom recurso visual para discutir possíveis melhorias durante uma reunião de *Post Mortem*.

Para criar um gráfico com a Taxa de Sucesso, você pode utilizar a consulta *PromQL* a seguir. Este é um exemplo de como obter a Taxa de Sucesso de um serviço que emite a métrica Prometheus `http_server_duration_count` :

```
100 * (1 - ( (sum(rate(http_server_duration_count{service_name="meu-servico",deployment_environment="prod",http_status_code=~"5.+"}[5m]))) or vector(0) / sum(rate(http_server_duration_count{service_name="meu-servico",deployment_environment="prod",http_status_code=~".+"}[5m])) ))
```

Você pode utilizar também a equação alternativa: `100 * (número de requisições bem-sucedidas / número total de requisições )` , como nesta consulta a seguir, que utiliza métricas Graphite. Operações aritméticas com métricas Graphite não são tão simples quanto em PromQL, portanto temos que usar funções. A divisão é executada pela função `divideSeries()` , e o numerador é a diferença, obtida pela função `diffSeries()` , entre o número total de requisições e o número de erros (soma dos valores dos *datapoints* que correspondem a ocorrências de respostas HTTP com código `5*`), e , finalmente, o resultado é multiplicado por 100 através da função `scale()` :

```
scale(divideSeries(diffSeries(sumSeries(xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.*:sum), transformNull(sumSeries(xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.5*:sum), 0)), sumSeries(xyz.loja-api.datacenter-a2.producao.trafego.http.respostas.*:sum)), 100)
```

Para explorarmos ainda mais a Taxa de Sucesso, vamos revisitar o SLI descrito anteriormente para o Website X e reescrevê-lo com esta nova terminologia e com uma nova condição:

- A Taxa de Sucesso do website não deve ficar abaixo de 99.99 %.

Isso significa que, ao observar a métrica de código de respostas HTTP emitidas em um minuto específico, o número de *datapoints* que correspondem a HTTP 5XX's (erros) não deve exceder 0.01 % da soma de todos os *datapoints*. Se expressarmos o SLO novamente, vamos encontrar novamente uma condição de nível de serviço (SLI) cercada por uma condição de frequência (SLO):

- 99.9 % do tempo, a Taxa de Sucesso do website não deve ficar abaixo de 99.99 %.

Repare que, no exemplo original, o SLI era bem mais restrito, pois não permitia nenhum percentual de erro (`Código de resposta HTTP != 5xx`). Porém, o SLI não precisa ser limitado a uma condição binária como essa, se o serviço produz erros ocasionais e possui uma operação satisfatória sob o ponto de vista de seus clientes, ou técnicas de resiliência como repetição de requisições (*retries*), isso quer dizer que seu SLI não precisa ser tão rígido, e que o serviço pode produzir erros e mesmo assim não acumular minutos ruins durante a observação do SLO.

P99 da Latência

Outro SLI na nossa lista de recomendações é o P99 da Latência, ou seja, o valor do nonagésimo nono percentil (do inglês: *99th percentile*) de latência. Como discutido no capítulo sobre métricas (capítulo 6), o *percentil* é uma estatística baseada em faixas de valores distribuídos de forma crescente, do menor para o maior, e o 99° *percentil* (ou apenas P99) representa o valor que está no topo (1% superior) da lista. SREs são bastante familiarizados com o fato de que médias são enganosas, por isso, é recomendado observar o P99 da latência do serviço e evitar análises imprecisas ao tentar medir a performance que está sendo apresentada aos usuários.

Vamos entender isso melhor: se observarmos apenas o tempo médio de resposta de um serviço, não olharemos para a porção de clientes que podem estar enfrentando uma lentidão substancial, com a qual, dependendo de quão crítico seja tal serviço nas jornadas de usuário,

problemas de performance podem impactar o negócio da organização significativamente. Vamos imaginar uma aplicação que emite uma métrica de tempo de resposta e que, dentro de um minuto, processou 12 requisições resultando na seguinte lista de *datapoints* e seus respectivos valores de tempo de resposta em milissegundos:

```
[100, 100, 100, 2000, 500, 1200, 600, 100, 300, 800, 100, 100]
```

Com essa simples lista de valores, é fácil identificar o *datapoint* cujo tempo de resposta teve o pior resultado: `2000 ms`, ou seja, o usuário esperou 2 segundos para obter a resposta, o que, em muitos casos, é bastante frustrante. Vamos utilizar esta lista para mais experimentos e ilustrar diferentes cálculos estatísticos e como seus resultados podem facilitar uma análise melhor da performance do serviço.

> Nesta seção vamos executar pequenos trechos de código Python para ilustrar os resultados dos cálculos a seguir. Para instalar o Python, siga as instruções na seguinte página: https://www.python.org/downloads/. Você pode copiar o conteúdo de cada listagem a seguir em um arquivo `p99_test.py`, por exemplo, e executar o mesmo com o comando `python p99_test.py` ou apenas executar o comando `python` em seu terminal para iniciar o console de REPL (*Read, Evaluate, Print, and Loop*) e copiar e colar cada comando, executando linha por linha para ver os resultados.

Se calcularmos a *média* (*mean*) dos *datapoints* de tempo de resposta, utilizando a lista anterior:

```
import statistics

datapoints_tempo_de_resposta = [100, 100, 100, 100, 100, 100, 300, 500,
600, 800, 1200, 2000]
print(statistics.mean(datapoints_tempo_de_resposta))
```

Vamos obter o resultado: `500 ms`, que está "escondendo" o número mais alto da lista (`2000 ms`). Se considerarmos este resultado da média (P50) no contexto da observação da latência de um serviço (monitorando o tempo de resposta de um aplicativo móvel ou de um website, por exemplo), podemos erroneamente assumir que o serviço está respondendo às requisições em menos de um segundo e que os usuários estão satisfeitos. Porém, existem usuários que estão bastantes frustrados com a performance do serviço e isso pode passar desapercebido com uma má escolha de agregação estatística de *datapoints*.

Dependendo do volume de tráfego e da importância do serviço, a observação pode ficar ainda mais granular com outros *percentis* como P99.9 ou até mesmo P99.999. Basicamente, SREs devem educar outros membros da organização para que uma experiência de usuário ruim não seja negligenciada, portanto é uma prática comum utilizar o P99 Latência como um SLI.

Aqui estão os passos para se calcular o *percentil*:

1. Ordene a lista.
2. Isole os valores superiores à faixa do *percentil* escolhido.
3. Retorne o valor que está imediatamente antes de tal faixa.

Aqui está um código simples para ilustrar este cálculo:

```
datapoints_tempo_de_resposta = [100, 100, 100, 100, 100, 100, 300, 500,
600, 800, 1200, 2000]

def pct(nums, percentile):
  sortedout = sorted(nums)
  i = (len(nums) * (percentile/100)) - 1
  index = round(i)
  return sortedout[index]

if __name__ == '__main__':
  print("percentil 75 dos datapoints de tempo de resposta")
  print(pct(datapoints_tempo_de_resposta, 75))
```

```
  print("percentil 99 dos datapoints de tempo de resposta")
  print(pct(datapoints_tempo_de_resposta, 99))
```

Que produz o seguinte resultado:

```
percentil 75 dos datapoints de tempo de resposta
600
percentil 99 dos datapoints de tempo de resposta
2000
```

Essa análise nos informa que 25 % dos usuários estão experimentando uma latência que é maior ou igual a `600 ms` (de acordo com o P75), e 1 % dos usuários está lidando com `2 s` de tempo de resposta. Isso nos conta uma história muito diferente em relação aos resultados do cálculo da média.

Para métricas Graphite, você pode explorar a função `nPercentile` (https://graphite.readthedocs.io/en/latest/functions.html#graphite.render.functions.nPercentile) e, para métricas Prometheus, utilize histogramas como o `http_request_duration_seconds_bucket` em conjunto com a função `histogram_quantile` mencionado no capítulo sobre métricas (para mais informações, verifique a documentação oficial: https://prometheus.io/docs/practices/histograms/). Por exemplo:

```
histogram_quantile(0.99,
sum(rate(http_server_duration_bucket{service_name="meu-
servico",deployment_environment="dev"}[5m])) by (le))
```

Uma ressalva importante para engenheiros(as) que estão começando a trabalhar com *percentis* é: evite calcular a média de *percentis*. Em algum momento isso pode parecer uma alternativa válida para agregar os *datapoints* de métricas de tempo de resposta emitidos por um serviço que correspondem a diferentes operações de sua API ou de diferentes réplicas, mas fique ciente de que isso é uma má ideia. Aqui está o porquê: imagine um serviço cuja réplica 1 emite os seguintes *datapoints* para tempo de resposta:

```
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

O P90, neste caso, é `19`. Agora imagine que a réplica 2 do mesmo serviço emite uma outra série de *datapoints*:

```
[91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

Ao calcular o 90° *percentil* para essa segunda lista, obtemos o número `99`. Se calcularmos a média entre esses *percentis*: `(99 + 19) / 2 = 59`, isso nos informa que a média do P90 do tempo de resposta deste serviço é `59 ms`, o que está errado, pois os usuários que interagem com a réplica 2 estão experimentando uma latência bem maior, o que torna essa análise imprecisa. O time responsável pela Observabilidade deve configurar mecanismos para agregar os *datapoints* relacionados ao tempo de resposta em uma única série temporal para fornecer um cálculo de *percentis* mais precisos.

Os responsáveis pelo serviço podem emitir métricas para operações distintas como:

```
xyz.loja-api.datacenter-
a2.producao.trafego.http.tempo_de_resposta.operacaoA
xyz.loja-api.datacenter-
a2.producao.trafego.http.tempo_de_resposta.operacaoB
xyz.loja-api.datacenter-
a2.producao.trafego.http.tempo_de_resposta.operacaoC
```

Mas é necessária a emissão de uma métrica que deve representar o tempo de resposta do serviço como um todo, como:

```
xyz.loja-api.datacenter-
agregado.producao.trafego.http.tempo_de_resposta_agregado
```

SRE deve incentivar simplicidade ao considerar estratégias de monitoramento, portanto observar o serviço sem distinguir operações específicas ou suas réplicas (ou nós de um *cluster*) é a maneira mais simples de se observar a performance de um serviço sem monitorar cada operação de forma granular e sem produzir análises ineficientes de performance. A **Lei de Hyrum** diz que, com

o tempo, todas as funcionalidades de um serviço serão instrumentadas por clientes, portanto não se deve observar partes específicas de um serviço e sim o serviço como um todo (mais detalhes em: https://www.hyrumslaw.com/). Uma forma comum de monitoramento que se aplica a múltiplos serviços ajudam a organização a implementar SRE de forma mais eficiente, então SREs devem promover mais padronização para facilitar o monitoramento e suporte de todos os serviços da organização.

9.3 Orçamento de Erro

O Orçamento de Erro (ou *Error Budget*) é uma maneira de expressar o SLO sob a ótica do número de eventos que podem ocorrer sem violar o acordo e, como o nome já expressa, funciona como uma "carteira" cujo saldo e gastos servem para orientar decisões de engenharia ou, mais especificamente, decisões relacionadas à necessidade de alocação de recursos destinados a confiabilidade. Se considerarmos um SLO de 99.9 % para um determinado serviço, isso quer dizer que o seu orçamento de erro é de 0.1%, ou seja, se a observação do serviço é baseada em minutos bons versus minutos ruins que ocorrem na janela de 1 semana, para encontrar o número total de minutos podemos calcular `60 * 24 * 7 = 10080` (pois 60 minutos = 1 hora, 24 horas = 1 dia e 7 dias = 1 semana), e como 0.1% de `10080` é `10.08` (0.08 minutos convertidos em segundos, 0.08 * 60, resulta em 4.8), isso quer dizer que o orçamento de erro para este serviço é de 10 minutos e 4.8 segundos ou, para simplificar, 10 minutos ruins em 1 semana.

Cada minuto é avaliado como bom ou ruim de acordo com os SLIs, e a frequência em que todos os SLIs devem ser atendidos é definida através do SLO e, finalmente, o número de minutos ruins que o serviço pode acomodar representa o seu orçamento de erro. Para ajudar a visualizar os conceitos discutidos neste capítulo, podemos usar como referência a "Pilha de Confiabilidade", ou *Reliability Stack*,

que é apresentada no livro *Implementing Service Level Objectives* (Hidalgo, 2020):

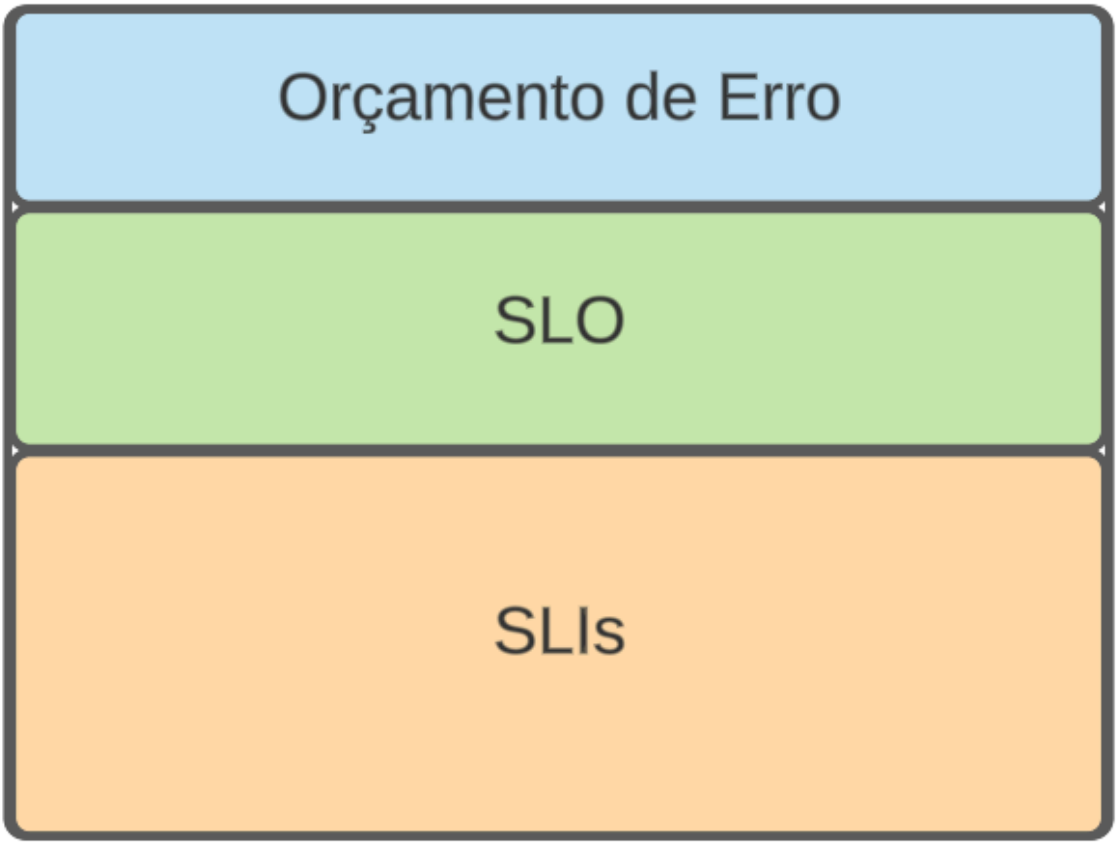

Figura 9.2: Pilha de Confiabilidade

Como você pode ver, os SLIs estão na base da estratégia de observação da confiabilidade de um serviço, pois devemos primeiro definir as condições para determinar se o serviço está operando de maneira satisfatória; depois, definimos a frequência em que esperamos que estas condições sejam atendidas através do SLO; finalmente, calculamos o Orçamento de Erro a partir do SLO para acompanhar se o serviço está mantendo o seu "saldo" sobre controle ou se está gastando mais do que deve.

Um detalhe muito importante sobre este orçamento é que *ele foi feito para ser usado!* Um time de desenvolvimento muito

conservador pode perder muito tempo verificando possíveis regressões em ambientes de Pré-produção e desacelerar o lançamento de uma determinada funcionalidade que torna o produto mais competitivo no mercado. Engenheiros(as) responsáveis por um serviço podem decidir verificar um grande número de configurações de infraestrutura e aplicar muito escrutínio com uma infinidade de testes para uma nova *feature*, porém, o melhor ambiente para obter *feedback* em termos de satisfação do usuário é o ambiente de Produção.

A execução de testes em outros ambientes como *Stage* ou Pré-produção proporcionam alguma evidência de que a funcionalidade está respondendo de acordo com as requisições, mas sempre existem discrepâncias entre tais ambientes de testes e aqueles que são expostos aos usuários reais. Pode ser uma pequena diferença em termos de infraestrutura que modifica o comportamento do serviço, alguma transação contendo um conjunto de dados formatados de uma maneira que não foi testada antes, ou até mesmo dependências como serviços de terceiros mantidos por entidades fora da organização que apresentam problemas intermitentes. Existe uma infinidade de anomalias que podem impactar um serviço e a real confiabilidade só pode ser testada de forma abrangente em Produção.

Com a estratégia certa, utilizando processos de mudança e *release* bem formulados, com mecanismos para reverter o *deployment* do novo código e desfazer mudanças (*rollback*), estes times devem tirar proveito do orçamento para, ocasionalmente, realizar *deployments* com um risco calculado e, caso algum imprevisto ocorra, podem restaurar o serviço para um estado saudável e discutir melhorias em seu processos de *design*, sua *pipeline* de testes, implementar mais métodos de resiliência e, desta forma, alcançar novos níveis de maturidade para seu serviço.

O raciocínio é o seguinte:

- Se o Orçamento de Erro não for utilizado, o time responsável pelo serviço não está ousando o suficiente e deve utilizar os minutos que tem a seu dispor.
- Se o Orçamento de Erro ficar negativo, isso significa que o time deve pensar em postergar algumas tarefas de desenvolvimento e focar mais em confiabilidade.

SREs devem educar times de desenvolvimento sobre o Orçamento de Erro para que eles possam tomar decisões estratégicas, baseadas nesta metodologia que busca um equilíbrio saudável entre manter a melhor experiência dos usuários, trazer mais eficiência que resulta em competitividade de mercado e a qualidade de vida do time de engenharia responsável pelo serviço. Vamos discutir mais utilizações práticas do Orçamento de Erro no próximo capítulo.

Capítulo 10
Boas práticas para alertas

Para cumprir a sua missão de "manter as luzes acesas", trabalhando para minimizar o tempo de indisponibilidade ou performance indesejada de algum serviço (sistema, website, aplicativo etc.), engenheiros(as) de confiabilidade de sites geralmente são aqueles(as) que ficam de plantão (*on-call*), carregando um *pager* que recebe as notificações de quaisquer eventuais alertas.

É comum que SREs fiquem no primeiro nível de contato de alertas de alta prioridade pois, uma vez que realizam uma investigação, também conhecida como "triagem", se tais alertas resultarem na criação de um incidente de acordo com o impacto da degradação de um ou mais serviços, SREs devem exercer o cargo de "comandantes de incidentes" (*Incident Commander* ou apenas IC) para coordenar outros times de engenharia, que devem trabalhar juntos para se certificar de que o sistema volte a operar dentro do SLA ou SLO, restaurando a experiência dos clientes o mais rápido possível.

Alertas auxiliam a engenharia com a detecção de problemas sem que seja necessário um monitoramento humano constante. O monitoramento ativo de serviços críticos e sua infraestrutura através de uma "Central de Comando", com engenheiros observando a telemetria proativamente 24/7, não é viável para a maioria das organizações, pois exige um investimento em centros geograficamente distribuídos onde engenheiros podem revezar a observação das métricas em relação ao SLA durante o horário de trabalho de cada localidade. Por exemplo, uma pessoa na costa oeste dos Estados Unidos, ao finalizar seu turno, é substituída no revezamento por alguém na Índia que inicia sua jornada de trabalho pela manhã e esta, por sua vez, quando concluir seu horário de trabalho, transfere a responsabilidade do monitoramento ativo para um(a) colega na Europa e, ao final do dia, algum engenheiro(a) na

costa oeste dos Estados Unidos toma seu lugar reiniciando o ciclo (este processo é conhecido como *follow the sun*). Este método é mais comum em grandes organizações que possuem usuários que também estão distribuídos globalmente, portanto, quando não é possível monitorar tudo proativamente e tempo todo, recorremos a alertas para que uma notificação possa ser produzida em caso de algum problema em potencial a qualquer hora do dia ou da noite.

Alertas, ou alarmes, podem ser configurados em serviços de monitoramento proprietários (como DataDog, New Relic, Splunk, Amazon Cloudwatch etc.) ou em soluções de código aberto (como Zabbix, Sensu, Prometheus, Grafana, entre outras). Alertas devem consultar alguma fonte de dados em uma determinada frequência e, de acordo com uma condição, enviar uma notificação para um ponto de contato ou algum outro serviço responsável pelo roteamento de notificações.

Existem muitas formas de notificações, ferramentas bastante utilizadas por engenheiros(as) *on-call* (como PagerDuty, OpsGenie e ServiceNow) podem enviar emails, SMS, mensagens via sistemas de mensageria (como Slack ou Microsoft Teams, por exemplo) ou iniciar uma chamada telefônica com uma gravação descrevendo brevemente os detalhes do alerta, permitindo que aqueles que recebem a chamada possam confirmar o recebimento do alerta (ação descrita em inglês como *acknowledgment*) para que fique registrado, de forma transparente para outros membros da organização, que algum(a) engenheiro(a) está ciente do chamado e que alguém está ativamente investigando aquilo que pode ser um incidente.

10.1 Alertando de acordo com SLOs e SLIs

Alertas são fundamentais para o monitoramento de serviços, já que são configurados para disparar notificações de acordo com alguma condição que indica algum problema em potencial que pode estar impactando usuários. As condições de um alerta são baseadas em algum sinal fornecido por um determinado serviço ou alguma dependência de infraestrutura (plataforma de *runtime*, banco de dados, fila de mensageria etc.). Esses sinais são geralmente expressados através de métricas, que são consultadas pelo sistema de alarme e têm seus *datapoints* avaliados de acordo com a condição, por exemplo: *tempo_de_resposta > 400ms*. Nem todo alerta é baseado em métricas, existem soluções que podem avaliar condições de alertas processando consultas em bancos de dados relacionais ou até mesmo identificando ocorrências de *logs* contendo algum padrão de texto. Mas para discutirmos boas práticas, vamos focar em métricas, já que este é o método mais comum para configuração de alertas.

A fadiga por alarmes

A fadiga por alarmes acontece quando um(a) engenheiro(a) recebe um grande número de alertas e, como não é fácil investigar cada instância de uma série de alertas que foram disparados em um determinado espaço de tempo, alguns são ignorados quando se percebe que o sistema está operando da forma desejada. Quando os alertas irrelevantes continuam produzindo notificações, engenheiros(as) tornam-se dessensibilizados(as) e passam a não dar tanta atenção aos sinais de possíveis problemas com o sistema, negligenciando ou postergando investigações e, desta forma, impactando a detecção de incidentes.

Problemas intermitentes ou transientes, que não têm uma duração tão substancial para quebrar o SLA de um serviço (cujo impacto é negligenciável), são conhecidos como *blips*, que é uma expressão utilizada para descrever fenômenos temporários ou insignificantes, como pontinhos que surgem temporariamente em um radar que representam uma breve mudança de algo que sai de seu estado

normal, mas que retorna à condição esperada rapidamente. Estes *blips* podem disparar alertas e contribuir para a fadiga por alarmes, então as configurações destes alertas devem ser ajustadas para não incomodar engenheiros(as) de plantão desnecessariamente.

Um alerta de baixa prioridade que ocorre durante um turno de plantão deve ser investigado e a causa raiz do problema deve ser resolvida ou, pelo menos, suficientemente mapeada para que alguma nota seja deixada para que o próximo(a) engenheiro(a) *on-call* possa tentar continuar a investigação no próximo turno. Caso isso ocorra fora do horário de trabalho ou durante o final de semana, SREs podem decidir silenciar o alerta até uma hora apropriada em que o time responsável pelo serviço estará disponível para auxiliar com uma investigação mais efetiva (se for possível esperar até então, dependendo da natureza do problema reportado pelo alerta, é claro).

Outro grande problema que ocorre em turnos de plantão são os alarmes falsos, pois estes falsos positivos consomem a energia daqueles que estão *on-call*. É bastante frustrante reagir ao *pager* e iniciar uma investigação observando métricas, *logs* e *traces* para, logo depois, descobrir que o serviço está operando como esperado e que o alerta disparou por engano, especialmente quando o *pager* acorda SREs no meio da noite interrompendo seu sono. Tal cenário pode ocorrer devido a uma série de fatores: problemas no banco de dados de série temporal utilizado pelo alarme causando ausência/ inconsistência nos *datapoints* das métricas, interrupção na emissão ou na coleta de métricas do serviço, alertas mal configurados etc. Falsos positivos são bastante prejudiciais e, se for apropriado, devem ser desabilitados para que sejam investigados depois em um momento apropriado. Quem está *on-call* no presente momento deve tentar evitar que tais alertas prejudiquem o próximo turno de seus colegas, criando um ciclo constante que melhora a qualidade do plantão de todos.

Quando o *pager* recebe muitos alertas, isso pode ser bem exaustivo, mas e se nenhum alerta ocorrer durante um grande período de

tempo? Quando tudo fica muito silencioso por muito tempo, alguma coisa pode estar errada. Isso pode acontecer se alguma dependência de seu sistema de alertas estiver indisponível, o que significa que problemas podem estar passando despercebidamente enquanto clientes estão encontrando erros ou enfrentando problemas de performance na aplicação (presumindo que, neste cenário, não exista um time de suporte que receba chamados para reportar problemas ao SRE *on-call*).

Para capturar problemas em seu sistema de alarme e se certificar de que as notificações estão fluindo como esperado, você pode configurar um alerta que deve ser disparado de propósito, a cada minuto, enviando uma notificação que deve ser recebida por uma aplicação que deve ser programada para acomodar dois cenários: se as notificações do alerta que dispara deliberadamente continuarem fluindo, a aplicação deve simplesmente registrar o alerta; porém, na ausência de notificações, o programa deve enviar alguma mensagem ao time de SRE (através de qualquer forma de contato alternativa que não seja o mesmo canal utilizado pelo sistema de alertas, como um email ou alguma mensagem diretamente enviada ao sistema de *chat* da organização). Esta técnica é conhecida, em inglês, como *Deadman's Switch*.

Alertas orientados por SLOS, SLIs e orçamentos de erro possuem uma estratégia que almeja reduzir a fadiga por alarmes. Vamos conhecer algumas dessas estratégias a seguir.

Cuidado com indicadores intermediários

Um serviço pode emitir uma infinidade de métricas: consumo de CPU, memória disponível, atividade de entrada e saída no disco (*Disk I/O*), número de nós de cluster, instâncias ou réplicas em operação, quantidade de tarefas sendo executadas em paralelo (*threads* ou *routines*), conexões ativas no banco de dados, atraso na entrega de mensagens, contagem de requisições HTTP etc. Se um alerta for provisionado para avaliar os *datapoints* de cada uma destas métricas, com um número significativo de serviços e componentes

de infraestrutura, o time de engenharia vai enfrentar uma sobrecarga de manutenção devido à grande quantidade de alertas. E boa parte destes alertas, ao disparar notificações, dificilmente vai realmente corresponder a um impacto real, ou seja, não devem indicar um problema que esteja afetando a experiência do usuário.

Imagine que um alerta é definido para notificar um(a) engenheiro(a) *on-call* quando o uso de CPU exceder 90%. Se tal alarme disparar e o serviço continuar operando como esperado, então esse limiar de uso computacional foi definido arbitrariamente sem uma análise real do comportamento da aplicação e não é uma indicação real de um problema. Portanto, SREs preferem uma estratégia de alerta que está mais orientada ao usuário e como ele percebe a performance da aplicação.

Os SLIs discutidos no capítulo anterior, Taxa de Sucesso e o P99 da Latência, são ótimos exemplos de indicadores que devem capturar anomalias que, de fato, estão impactando a experiência do usuário. Quando a medição de CPU ou memória de um servidor apresenta algum ponto que se aproxima da saturação e este cenário impacta a usabilidade do cliente, a aplicação geralmente produz erros em outro nível de abstração acima da camada do servidor ou sistema operacional, como no caso de serviços que operam como uma API HTTP, onde o cliente deve receber códigos de resposta do tipo 5XX, por exemplo, *500: Internal Server Error* ou *502: Bad Gateway* caso a aplicação entre em colapso e o balanceador de carga não conseguir rotear a requisição para um servidor HTTP. Portanto, esses sinais de alto consumo de CPU ou memória não são ideais para a observação de SLIs, já que são apenas indicadores de baixo nível que geralmente produzem erros para camadas superiores. Os erros que ocorrem no nível de servidor em relação à exaustão de recursos acabam produzindo erros no nível da aplicação, ou seja, são apenas indicadores intermediários.

No livro *Implementing Service Level Objectives* (Hidalgo, 2020), o autor descreve estes indicadores intermediários como *SLI Proxies*, pois encontram-se em uma camada em que, quando valores

anômalos são encontrados, nem sempre vão refletir em um impacto real sob a perspectiva do cliente. SREs devem ficar atentos a tais cenários e evitar alertas baseados em indicadores intermediários, especialmente para diminuir o número de alertas que podem ser disparados durante um turno de plantão.

Exemplo prático de um alerta de SLI usando Prometheus e Grafana

Para conduzirmos um experimento onde um alerta no Grafana avalia métricas do Prometheus, temos que criar uma simulação alimentando o Prometheus com *datapoints*. Isso pode ser feito através da ferramenta `promtool`, que possui comandos para adicionar entradas no banco de dados de série temporal. Para ilustrar o funcionamento de um alerta baseado na taxa de sucesso de um serviço, vamos utilizar o código abaixo para simular *datapoints* de uma métrica `http_requests_total` que serão distribuídos através de uma linha do tempo no passado.

Vamos primeiro introduzir instruções de *bash scripting* em um arquivo `gerador-de-datapoints-historicos.sh`. Copie o seguinte código para o arquivo:

```
#!/bin/bash

cat << EOF
# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total counter
EOF

# Data/Hora atual
AGORA=$(date +"%s")
# Numero de datapoints historicos
NUMERO_DE_MINUTOS=60
# As metricas vao comecar a surgir aqui
PONTO_DE_PARTIDA=$(($AGORA - ($NUMERO_DE_MINUTOS * 60)))
```

```
NUMERO_DE_ERROS=0
MINUTO_EM_QUE_ERROS_COMECAM_A_OCORRER=10

for i in $( seq 1 $NUMERO_DE_MINUTOS )
do
   PROXIMA_DATA=$(($PONTO_DE_PARTIDA + ($i * 60)))
   echo "http_requests_total{code=\"200\",service=\"servicoX\"} 1000
$PROXIMA_DATA"
   echo "http_requests_total{code=\"500\",service=\"servicoX\"}
$NUMERO_DE_ERROS $PROXIMA_DATA"
   if [[ $i -ge $MINUTO_EM_QUE_ERROS_COMECAM_A_OCORRER ]]; then
     # Entramos na janela de erros, vamos incrementar o num de erros a
cada minuto
     NUMERO_DE_ERROS=$((NUMERO_DE_ERROS+1))
   fi
done

echo "# EOF"
```

Esse script deve gerar métricas com uma contagem fixa de `1000` ocorrências de requisições cujas respostas produzem um código `200` e, a partir do minuto `10`, devem incrementar o número de erros (código de resposta `500`) a cada minuto até concluir suas `60` iterações, imprimindo métricas que, de acordo com o ponto de partida, vão simular a emissão de *datapoints* distribuídos na última hora. O script é executado com o comando: `bash gerador-de-datapoints-historicos.sh`.

Uma vez que as métricas forem geradas, inicie o contêiner Prometheus com o controle do terminal em uma sessão bash:

```
docker run --rm -it --name prometheus -p 9090:9090 --network sre-test-
network -v $(PWD)/prometheus.yml:/etc/prometheus/prometheus.yml --
entrypoint sh prom/prometheus
```

Copie e cole as métricas produzidas pelo script em um arquivo `metricas` (você pode utilizar um editor de texto dentro do contêiner: `vi metricas`). Execute o comando `promtool tsdb create-blocks-from`

`openmetrics metricas` para criar métricas no passado (processo conhecido como *backfilling*) e inicie o Prometheus com `/bin/prometheus --config.file=/etc/prometheus/prometheus.yml .`

Na interface de exploração do Grafana (`http://localhost:3000/explore`), você pode ver o resultado com a seguinte consulta *PromQL*:

```
100 * (1 - ( (sum(rate(http_requests_total{service="servicoX",code=~"5.+"}
[5m]))) or vector(0) /
sum(rate(http_requests_total{service="servicoX",code=~".+"}[5m])) ))
```

Aqui está uma imagem com a captura de tela do resultado:

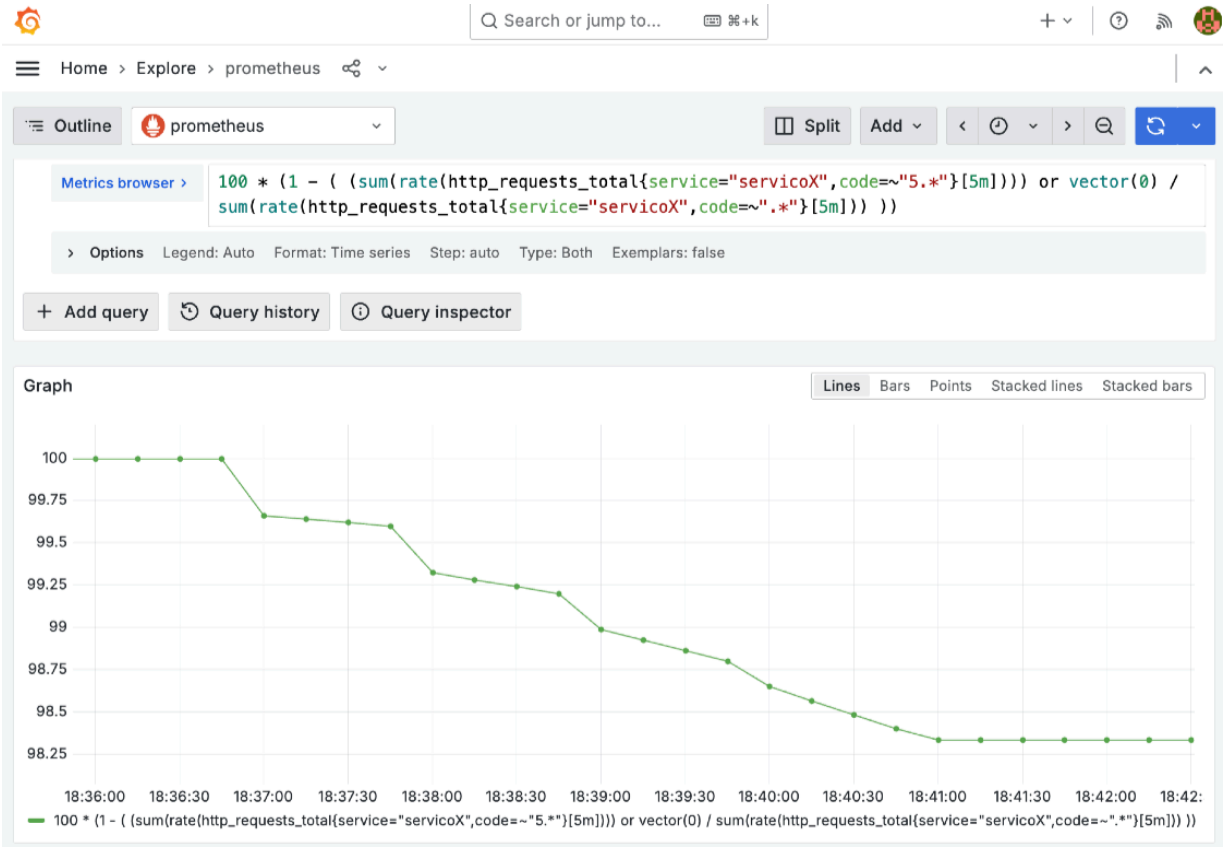

Figura 10.1: Taxa de sucesso em queda produzida por datapoints históricos criados manualmente

Agora que temos métricas que podem ser avaliadas pelo alerta, podemos criar uma regra de alerta (*alert rule*) no Grafana. Porém,

para que uma regra de alerta seja criada, existe um pré-requisito, que é o ponto de contato (*contact point*). Esta configuração tem que ser definida para que o alerta tenha um destino para sua notificação quando o alerta for disparado.

Com o contêiner Grafana ainda em execução, vamos criar o ponto de contato navegando para o console de *Contact points* do Grafana (`http://localhost:3000/alerting/notifications`) e clicando em "+ Add contact point". No formulário "Create contact point", note que o campo de seleção "Integration" possui muitas opções de integrações que podem processar a notificação do alerta (Email, Discord, PagerDuty, Slack etc.). Como estamos conduzindo um experimento simples isolado em uma máquina local, vamos escolher a opção *Webhook* para que o Grafana envie uma requisição HTTP para um determinado alvo quando o alerta disparar.

Para confirmar o recebimento da notificação do alerta, vamos utilizar um simples servidor HTTP que deve imprimir as requisições recebidas em seus *logs*. Inicie outra sessão do terminal e execute o seguinte contêiner Docker `daime/http-dump` em nossa rede de teste `sre-test-network` :

```
docker run --rm -it --name recebedor-de-alertas -p "8080:8080" --network sre-test-network daime/http-dump:latest
```

De volta ao console de criação do ponto de contato do Grafana, com a integração do tipo *Webhook* selecionada, preencha o campo *URL* com: `http://recebedor-de-alertas:8080` . Você pode testar o ponto de contato clicando no botão "Test" e enviando uma notificação predefinida ou customizada. Envie algumas notificações de teste, inspecione os *logs* produzidos pelo contêiner em seu terminal e salve a configuração no console do Grafana para criar um ponto de contato chamado "Recebedor de Alertas":

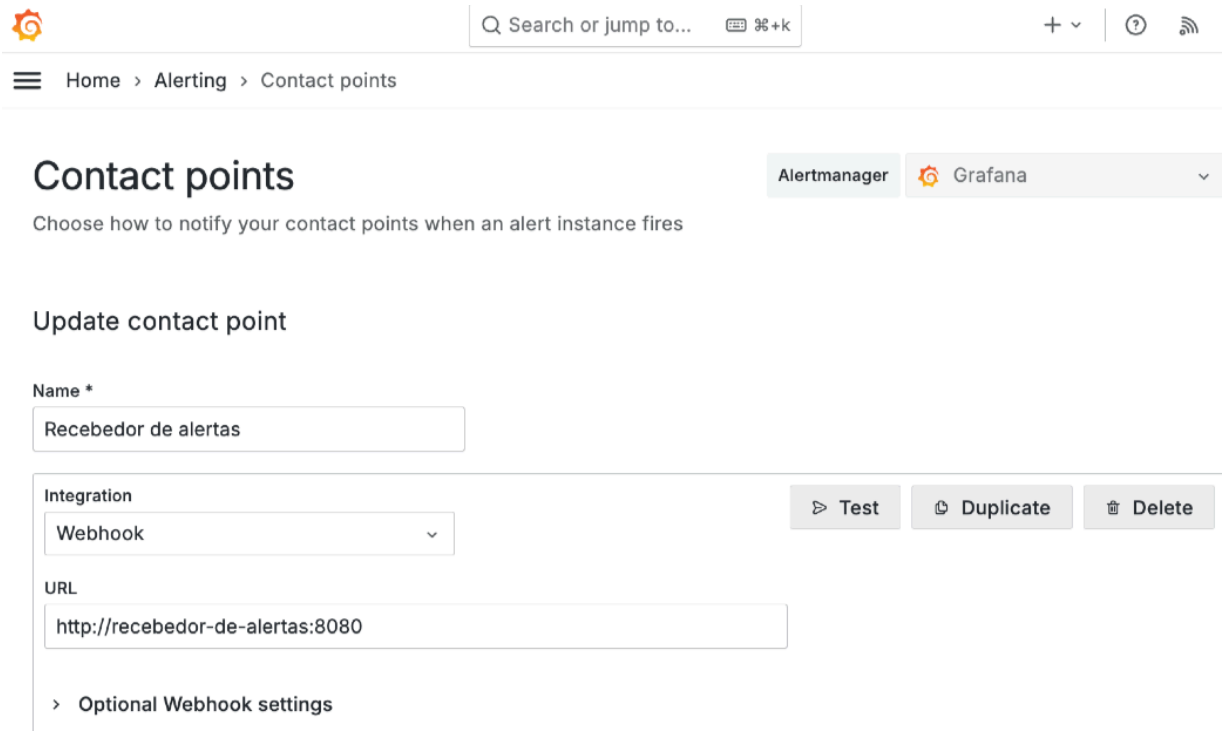

Figura 10.2: Criando um ponto de contato no Grafana

Agora podemos criar a nossa regra de alerta. Navegue para o console de listagem de regras de alertas (`http://localhost:3000/alerting/list`) e clique em "New alert rule", preencha o campo "Name" para definir o título deste alerta (exemplo: "servicoX SR SLI abaixo de 99.9") e proceda com a configuração da consulta e da condição do alerta (*query and alert condition*). A fonte de dados (*data source*) selecionada deve ser *Prometheus*, e a consulta que será avaliada pelo alerta deve conter a mesma instrução *PromQL* citada anteriormente, que corresponde à taxa de sucesso do serviço X.

Já que os *datapoints* que foram criados para esta simulação foram gerados por um script e não fazem parte de uma métrica que está sendo emitida constantemente, a avaliação da regra de alerta não vai encontrar novas entradas no banco de dados do Prometheus nos próximos minutos. Por isso, já que estamos conduzindo um experimento para ilustrar o funcionamento do alerta, vamos configurar a janela da consulta do alerta para avaliar os *datapoints*

presentes no passado e processar os valores da taxa de sucesso consultando um período entre 3 horas e 1 hora atrás (`From: now-3h` e `To: now-1h`).

> Você pode ajustar essa janela de acordo com a presença de métricas no Prometheus ou revisitar os passos para criar um novo conjunto de *datapoints*: reinicie o contêiner do Prometheus em outro terminal, execute o script `gerador-de-datapoints-historicos.sh` , copie as métricas para um arquivo `metricas` dentro do contêiner, execute a ferramenta `promtool` para criar os datapoints e inicie o Prometheus novamente.
>
> Para configurar um alerta em uma situação da vida real, você deve considerar uma janela mais próxima do tempo presente para capturar problemas da forma mais rápida possível (por exemplo: `From: now-5m` e `To: now`), mas sempre ajustando o período da consulta de acordo com as peculiaridades de cada serviço, considerando a frequência em que as métricas são emitidas e a presença dos *datapoints* no banco de dados de série temporal.

Aqui está uma captura de tela ilustrando como a janela da consulta do alerta pode ser definida:

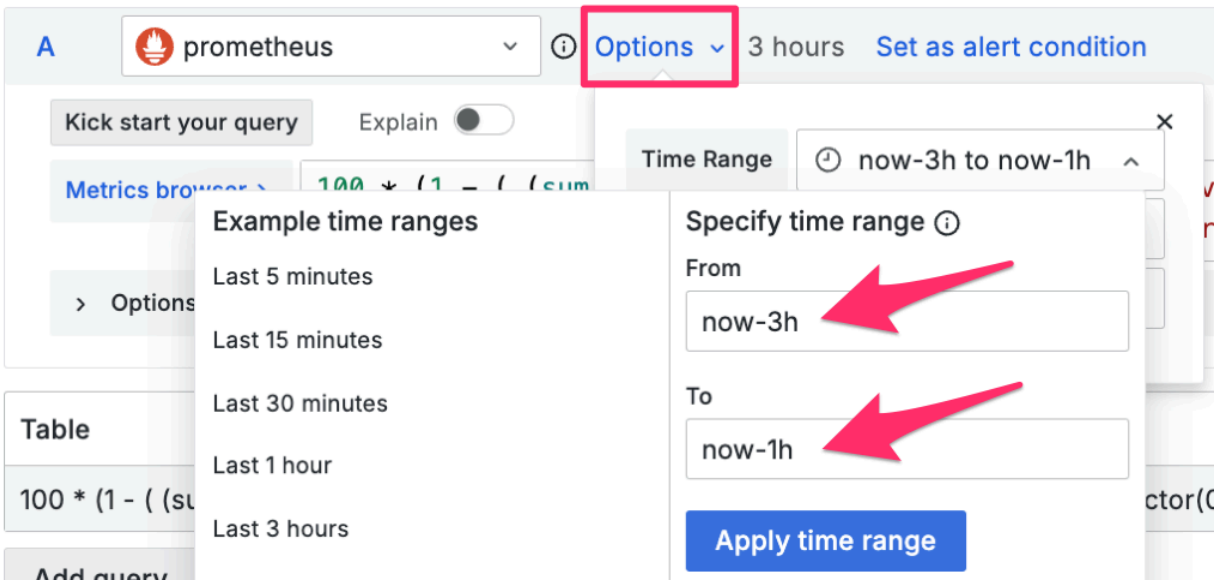

Figura 10.3: Definindo janela da consulta da regra de alerta

Uma vez que a consulta for definida, crie a condição do tipo "limiar" (do inglês, *Threshold*) cuja entrada (*Input*) será a consulta "A" (primeira e única consulta do alerta) e informe o critério que será avaliado pelo alerta que, neste caso, como queremos que o alerta seja disparado quando o SLO for violado, então a condição será `taxa de sucesso < 99.9`, portanto selecionamos a opção "IS BELOW" que significa "está abaixo de" e preenchemos o próximo campo de texto com `99.9`. Clique em "Preview" para pré-visualizar a avaliação do alerta e verificar se a condição será encontrada ou não. O estado do alerta deve mudar para "FIRING", que quer dizer que o mesmo vai disparar pois nossa simulação de *datapoints* contém erros suficientes para consumir mais de 0.1 % do orçamento de erro.

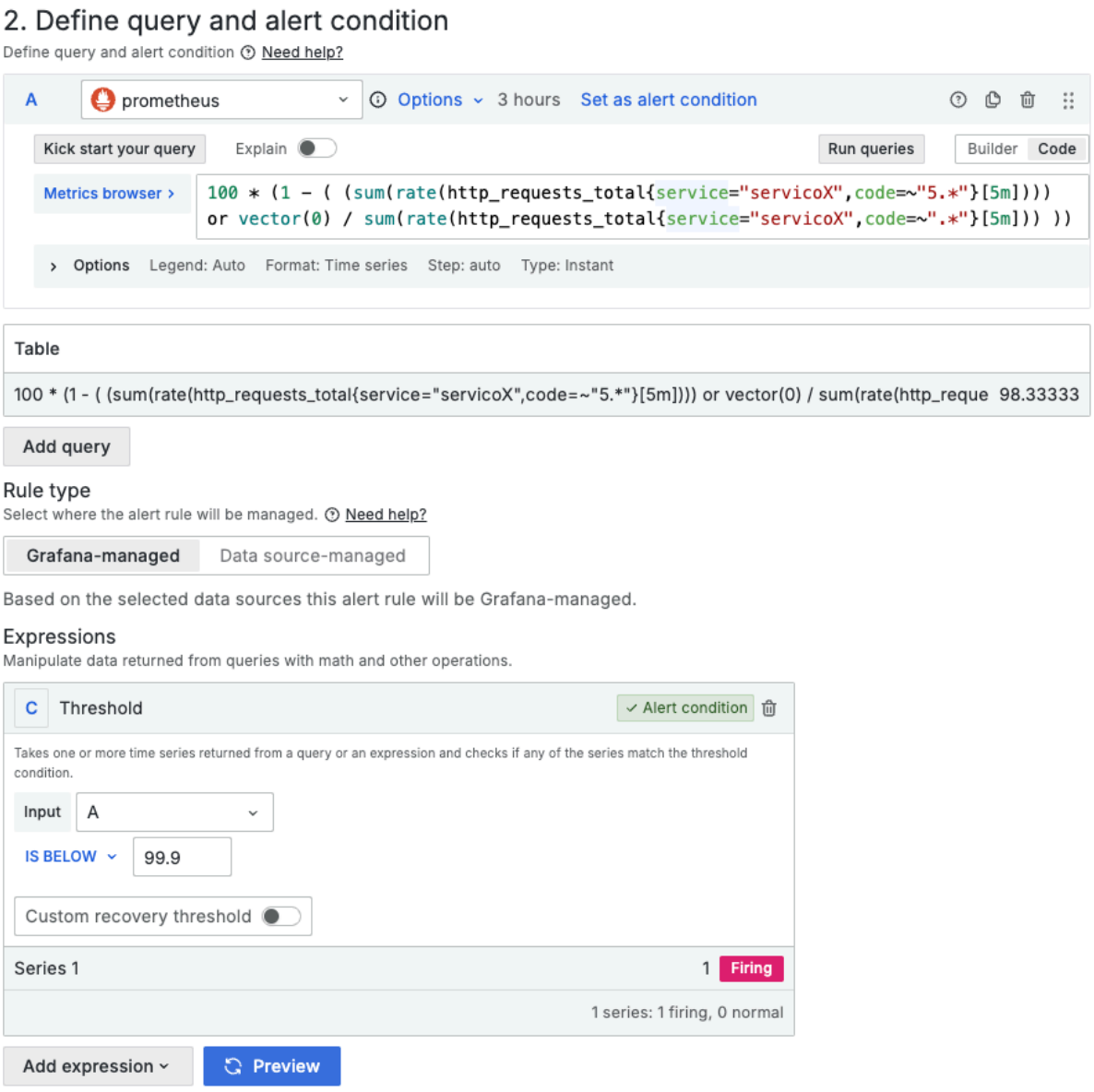

Figura 10.4: Criando uma regra de alerta baseado em um SLI

Ainda temos alguns passos a serem tomados para concluir o experimento:

- Crie uma pasta lógica (*Folder*) que deve abrigar o alerta. Fique à vontade para usar qualquer nome, como "Testes SRE", por exemplo.
- Configure um grupo de avaliação para informar o intervalo em que o alerta será avaliado (*Evaluation group and interval*), esta

será a frequência em que o alerta vai executar a consulta e verificar se o resultado cumpre a condição para disparar o alerta, vamos configurar um intervalo de 1 minuto.

- Selecione o período em que o alerta deve permanecer dentro da condição para que seja disparado (*Pending Period*). Mesmo que a avaliação do alerta, em um determinado minuto, cumpra o critério especificado em sua condição, os operadores (as) podem decidir se o alerta deve ser disparado imediatamente naquele minuto, ou apenas se a condição for cumprida em um número arbitrário de minutos contínuos. Para este experimento, vamos selecionar a opção 5 minutos, já que, para um SLO de 99.9% observado em uma janela de 1 semana (com 10 minutos ruins permitidos), 5 minutos representam uma queima de 50% do orçamento de erro, portanto é uma configuração comum para este tipo de alerta baseado em SLIs.

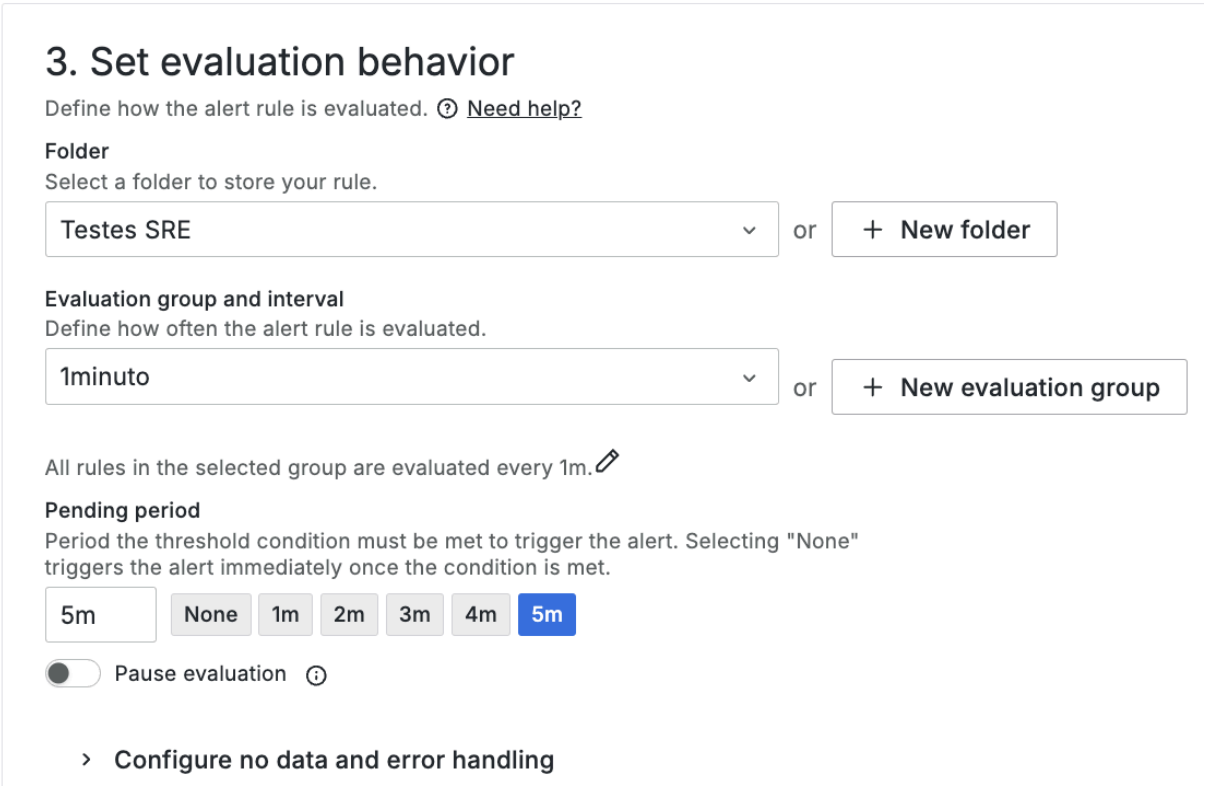

Figura 10.5: Definindo quando a regra de alerta deve disparar

O último passo a ser tomado é a definição do ponto de contato desta regra de alerta, onde podemos selecionar o nosso "Recebedor de Alertas" definido anteriormente.

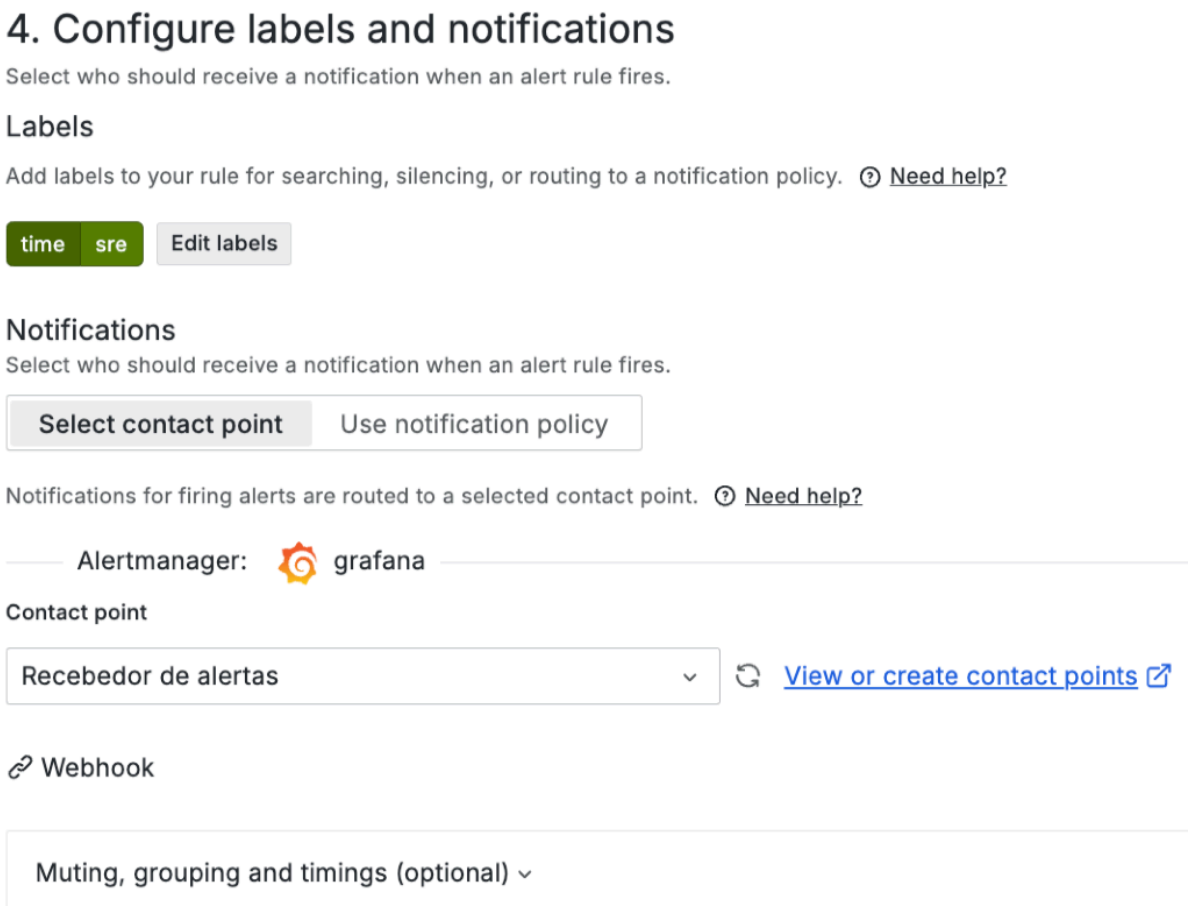

Figura 10.6: Selecionando o ponto de contato da regra de alerta

Após definir o ponto de contato, vá para o topo da página e clique em "Save rule and exit" no canto superior direito, para finalizar a criação do alerta.

Agora observe o alerta em operação na página de listagem de regras de alerta (`http://localhost:3000/alerting/list`). Note que, ao salvar a regra de alerta, o Grafana vai iniciar um novo processo que vai executar a expressão definida na consulta do alerta, dentro da frequência de sua avaliação (neste caso: A cada minuto). O alerta deve transicionar para o estado de pendência (*PENDING*) quando

sua próxima avaliação acusar que o valor da taxa de sucesso está abaixo do limiar `99.9`, e depois de 5 minutos (de acordo com o período de pendência definido anteriormente), deve mudar novamente para o estado de disparo (*FIRING*). Aqui está outra captura de tela para ilustrar essas transições de estado:

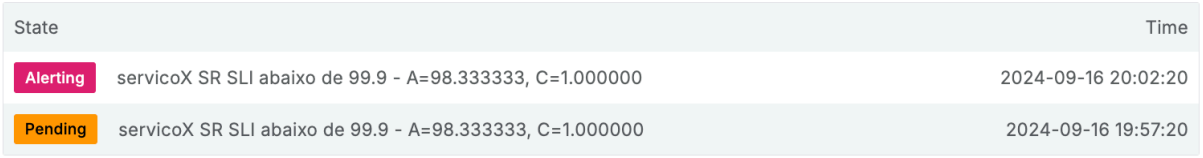

Figura 10.7: Transições de estado da regra de alerta

E então, finalmente, o nosso "Recebedor de Alertas" deve gerar uma nova entrada de *log* contendo um JSON similar a este que se encontra a seguir:

```
{
  "title": "[FIRING:1] servicoX SR SLI abaixo de 99.9 Testes SRE (sre)",
  "receiver": "Recebedor de alertas",
  "status": "firing",
  "alerts": [
    {
      "status": "firing",
      "labels": {
        "alertname": "servicoX SR SLI abaixo de 99.9",
        "grafana_folder": "Testes SRE",
        "team": "sre"
      },
      "annotations": {},
      "startsAt": "2024-09-16T18:02:20Z",
      "endsAt": "0001-01-01T00:00:00Z",
...
      "values": {
        "A": 98.33333333333333,
        "C": 1
      }
    }
  ],
```

```
...
}
```

A regra de alerta vai continuar executando seus ciclos de avaliação a cada minuto. Se deixarmos este contêiner do Grafana em execução por mais de 1 hora, o alerta vai, eventualmente, reportar a ausência de dados (disparando uma notificação devido à transição para o estado *No Data*). Isso vai acontecer quando o Prometheus não retornar novos *datapoints* nas consultas futuras, então você pode configurar a regra de alerta para ignorar tais casos e evitar que o alerta dispare. Porém, em situações da vida real, quando um serviço interrompe a emissão de suas métricas, isso pode ser bastante crítico pois uma organização não pode ficar "às cegas" quando se trata do monitoramento de seus serviços e seus respectivos SLOs.

Para reproduzir esse experimento configurando uma janela de observação no tempo presente (sem utilizar uma simulação de *datapoints* do passado), com um fluxo constante de novos *datapoints*, você pode revisitar a aplicação de exemplo e escrever uma alguma lógica para produzir métricas a cada minuto simulando erros para observar a avaliação desta métrica em um alerta.

Alertando com o orçamento de erro

Os alertas baseados no orçamento de erro utilizam cálculos entre as violações de SLO e sua janela de observação para identificar cenários em que o serviço indica um comportamento que vai, eventualmente, esgotar seu orçamento. Esse alerta pode ser disparado em caso de erros contínuos, que ocorrem em uma frequência consistente resultando na utilização de todo o orçamento de erro em um curto espaço de tempo, ou quando o SLO é violado esporadicamente, com eventos ruins ocorrendo em momentos separados, acumulando violações suficientes para que seja identificada uma tendência à exaustão do número de eventos ruins que são permitidos antes do fim de sua janela de observação.

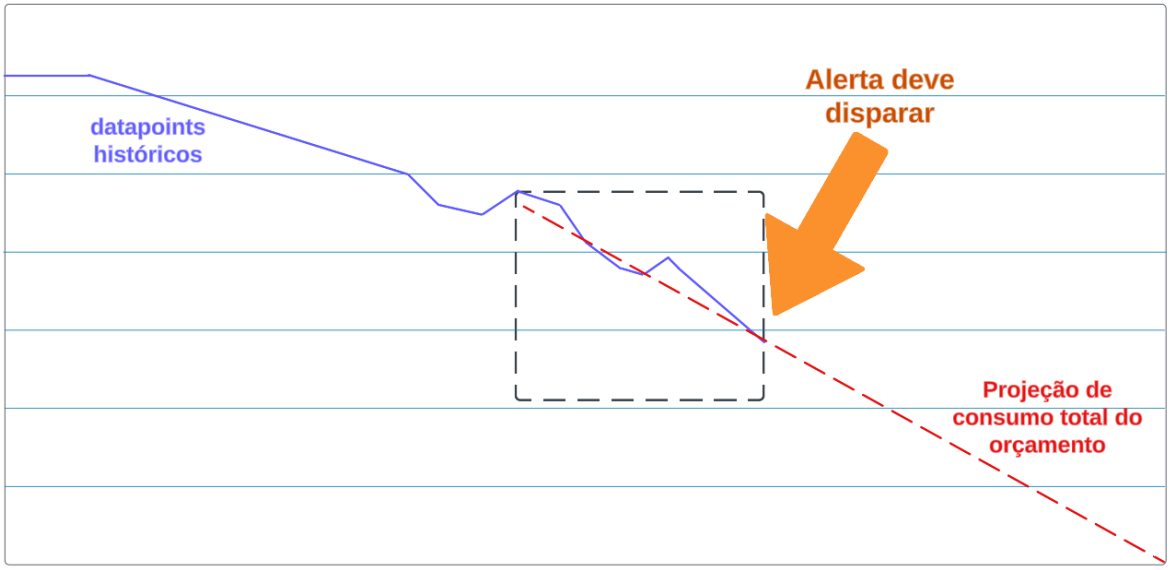

Figura 10.8: Previsão de exaustão do orçamento de erro

Este tipo de alerta é baseado na taxa de queima (do inglês, *burn rate*), que é usada para determinar a velocidade em que o orçamento de erro será consumido até exaustão. A taxa de queima é a proporção entre os erros (minutos ruins ou contagem de eventos ruins) e a quantidade de erros permitidos de acordo com o SLO, observada em múltiplas janelas de tempo menores do que o período de observação do SLO. Se um SLO for observado em uma janela de 28 dias, a taxa de queima pode ser avaliada a cada 5 minutos, 10 minutos ou mesmo a cada X horas, por exemplo.

O livro *Site Reliability Workbook* (Murphy, 2018), no capítulo "Alertando por SLOs", contém uma análise da taxa de queima em relação à taxa de erro e ao tempo que se leva para consumir todo o orçamento. A tabela a seguir ilustra como a taxa de queima pode ser utilizada para determinar os critérios de um alerta, pois à medida que ela é incrementada, o orçamento é consumido mais rapidamente. Quando a taxa é constante (1), isso significa que o orçamento está sendo consumido exatamente no limiar de seu SLO (taxa de erro 0.1%), o que levaria toda a janela de observação, neste caso 30 dias, para que o orçamento seja consumido por

completo. Se a taxa de queima dobrar, o consumo do orçamento também será acelerado, atingindo a exaustão na metade do tempo (15 dias):

Taxa de queima	Taxa de erro com SLO de 99,9%	Tempo até o consumo total
1	0.1%	30 dias
2	0.2%	15 dias
10	1%	3 dias
1000	100%	43 minutos

Existem muitas abordagens que podem ser tomadas quando se trata de alertas baseados na taxa de queima. Observar uma janela curta, apesar de possuir um bom tempo de detecção, pode resultar em alertas desnecessários caso o serviço volte a operar normalmente depois de um curto período de degradação, uma janela longa pode apresentar uma precisão melhor, mas exige um tempo de espera significativo para que o alerta seja restaurado (*reset time*) e pare de disparar. Devido a estas ineficiências encontradas em cada abordagem isolada, a prática recomendada é a avaliação de múltiplas janelas e múltiplos limiares da taxa de queima.

Para abstrair a complexidade da configuração de tais alertas, sugerimos a utilização de ferramentas como *Sloth* (https://sloth.dev/), que geram todas as configurações necessárias para provisionar tais alertas através de um manifesto, também conhecido como "arquivo de especificação de SLO" (*SLO spec*). Estes arquivos de configuração são populados com informações como `objective` (SLO), seus SLIs e suas respectivas métricas (`error_query` e `total_query`), assim como as configurações referentes aos alertas (metadados como `labels` e `annotations`) e os alvos das notificações para *pages* e *tickets* (`page_alert` e `ticket_alert`).

10.2 Alertas e níveis de prioridade

Alertas devem ser configurados para notificar times de engenharia ou criar tickets de acordo com a prioridade apropriada. Lembre-se: "Quando tudo é importante, então nada é realmente importante", ou seja, nem todo serviço possui funções que estão associadas com as funcionalidades mais críticas do produto. Vamos considerar, por exemplo, dois sistemas de uma companhia aérea: o primeiro é utilizado pelos clientes para comprar passagens e selecionar seus assentos e o segundo processa pedidos do serviço de bordo, oferecendo a consulta de um cardápio e o pagamento de refeições e bebidas. Ambos são bastante utilizados pelos usuários, mas fica bem claro qual serviço deve estar associado a um nível de prioridade mais baixa: um sistema de serviço de bordo indisponível pode frustrar alguns usuários, mas os próprios membros da tripulação podem atender aos clientes processando pedidos manualmente para contornar o sistema fora do ar. Porém, quando usuários não podem comprar passagens através do website da companhia aérea, cada minuto em que voos não são agendados causa um grande impacto financeiro para a empresa.

Por isso, os serviços devem ser classificados em camadas de prioridades, conhecidas, em inglês como *tiers*. Um serviço *tier* 1 está associado com as partes mais importantes de um produto (website, API, aplicativo móvel etc.), aquelas funcionalidades que estão diretamente alinhadas com o negócio da organização, como o que representa a venda de passagens para uma companhia aérea. Outros sistemas cuja indisponibilidade não causa um impacto tão severo ou cujas funcionalidades podem ser temporariamente ignoradas ou contornadas são considerados *tier* 2. Existem também serviços *tier* 0 (camada zero), que são categorizados como de altíssima prioridade pois, quando a base de tudo é afetada, então todo o resto é afetado. A infraestrutura que abriga outros serviços, *data centers* ou soluções na nuvem (*PaaS*), são exemplos de

serviços de camada zero, já que as aplicações não têm como executar seu código sem os servidores, máquinas virtuais, plataformas de *runtime*, rede etc.

Outros serviços que podem ser considerados camada zero estão associados com Observabilidade, como os sistemas responsáveis por métricas, *logs* e *traces*, por exemplo. Isso pode dividir opiniões, pois algumas pessoas podem considerar que não tem problema enfrentar um "apagão" de monitoramento se o cliente estiver feliz, interagindo com serviços normalmente, porém, como podemos saber ao certo? Quando não é possível observar a performance e o status de serviços críticos, pode existir uma quantidade considerável de usuários encontrando erros ou respostas de alta latência e engenheiros(as) responsáveis jamais saberão se o SLO está sendo mantido. Tais níveis de prioridade de serviços podem ser especificados em um catálogo de serviços, que é uma ferramenta essencial para SREs e será explorada no próximo capítulo.

A definição de níveis de prioridade de serviço é um passo importante para tornar um serviço elegível para suporte de SREs, ou seja, se engenheiros(as) vão ficar de plantão e reagir de acordo com alertas que vão notificar o *pager*, a triagem de tais alertas será conduzida de acordo com a prioridade associada com o serviço. Alguma violação de SLO que dura 10 minutos para um serviço *tier* 2 talvez justifique um incidente de severidade 3 e pode ser escalado para severidade 2 se não for mitigado em até 30 minutos, enquanto um serviço *tier* 1 que violou o SLO por 5 minutos, uma vez que a triagem confirmar que não é um falso positivo e que causa de fato um impacto significante, pode justificar um incidente de severidade 2 imediatamente e tornar-se um severidade 1 em poucos minutos, principalmente se a degradação do sistema ficar mais intensa.

SREs podem tomar estas decisões em colaboração com gerentes e especialistas dos serviços impactados, já que possuem conhecimento das capacidades funcionais do sistema e de como a experiência de usuário é prejudicada por algum comportamento indesejado ou indisponibilidade (quando o sistema está *down*). Outra opção, que

está em um nível de amadurecimento de processos mais elevado, é a formalização de uma *tabela de definição de incidentes* para cada serviço, mas tal tabela deve ser definida colaborativamente para que todas as entidades envolvidas estejam cientes do procedimento em caso de algum incidente.

Aqui está um exemplo para ilustrar:

Serviço	Tier	Sev0	Sev1	Sev2	Sev3
AWS VPC/EC2	0	Down > 30m	Down > 15m	Down > 10m	Down > 5m
Passagens	1	SR < 99.95 por 1h	SR < 99.95 > 15m	SR < 99.95 > 10m	SR < 99.9 > 10m
Menu	2	N/A	SR < 99 > 2h	SR < 99 > 1h	SR < 99 > 15m

Veja que nesta tabela temos serviços seguidos dos respectivos *tiers* e o critério de cada uma das possíveis severidades de um incidente. O primeiro exemplo (AWS VPC/EC2) compreende a infraestrutura que abriga todos os outros serviços, a *Virtual Private Cloud* (VPC) representa a rede que estabelece a comunicação com as aplicações e o *Elastic Compute Cloud* (EC2) é o serviço de máquinas virtuais que é utilizado pelas plataformas de *runtime* para executá-las, portanto, este primeiro serviço é classificado como *tier 0* (prioridade máxima). De acordo com a tabela: se ficar indisponível por mais de 5 minutos um incidente de severidade 3 deve ser iniciado, se continuar indisponível por mais de 10 minutos, deve-se escalar o incidente para severidade 2 e, da mesma forma, severidade 1 case haja uma indisponibilidade com uma duração de mais de 15 minutos e, finalmente, severidade zero se o ambiente de produção estiver inoperável por mais de 30 minutos.

A tabela mostra dois outros serviços: "svcPassagens" e "svcMenu", seguindo o exemplo mencionado anteriormente envolvendo o sistema de uma companhia aérea, cada um dos serviços também possui o seu respectivo *tier* e os critérios para definir as diferentes severidades de um incidente, porém, desta vez, em vez de definir condições relacionadas a indisponibilidade, os critérios para cada severidade estão relacionados com a taxa de sucesso (*Success Rate*

ou apenas SR, como visto na tabela). Isso quer dizer que para o serviço "svcPassagens", responsável por compras de passagens, se a taxa de sucesso ficar abaixo de `99.95%` por mais de 15 minutos, este cenário problemático está elegivel para um incidente de severidade 1.

Categorizar serviços de acordo com seus níveis de prioridade pode ser desafiador devido à divergência de opiniões, já que engenheiros(as) podem ter perspectivas diferentes quando se trata de funcionalidades de serviços e como a degradação das mesmas impacta clientes. Para conduzir tais discussões e formalizar *tiers* para cada serviço suficientemente relevante, SREs podem utilizar as ressalvas mencionadas anteriormente para estabelecer o tom da reunião e ajudar todos os envolvidos a tomar tais decisões.

UTILIZANDO POKER PARA CLASSIFICAR O *TIER*

Para lidar com disputas, os membros do time responsáveis pelo serviço podem tentar um jogo de "Poker", que é inspirado em uma atividade do *Scrum* (metodologia ágil) para escolher os "pontos de esforço" de tarefas. O jogo funciona da seguinte maneira: todos escolhem suas cartas silenciosamente, escondendo a carta selecionada com o respectivo número da camada do serviço (de acordo com suas próprias opiniões) e, uma vez que todos escolherem seus números, devem virar suas cartas ao mesmo tempo revelando as escolhas. Se existir uma grande maioria que tem preferência por um determinado número, então registra-se aquela escolha predominante como o *tier* para aquele determinado serviço; se existir alguma escolha que seja bastante divergente, todos devem ouvir a opinião daquela pessoa e, dependendo da discussão, mudar a escolha do nível de prioridade do serviço em questão. Geralmente, participantes que selecionam o maior e aqueles que selecionam o menor número são aqueles que devem discutir entre si para justificar suas escolhas e expor suas opiniões. Através dessa discussão, às vezes são revelados detalhes que não tinham sido considerados por outros, que podem acabar persuadindo outros a aceitarem a escolha de um outro nível de prioridade.

10.3 Documentando passos acionáveis para alertas

Uma prática comum para oficializar o fornecimento de suporte de serviços é a criação de um documento que ajuda SREs a

aprenderem o mínimo necessário sobre um serviço para saberem com o que estão lidando em caso de algum evento disruptivo que pode impactar clientes, tais documentos são conhecidos como *runbooks* ou *playbooks*. Estas expressões em inglês remetem-se a pequenos caderninhos que treinadores esportivos usam para documentar as jogadas ensaiadas da equipe, descrevendo o que cada membro do time deve fazer para serem bem-sucedidos na partida enquanto seguem um plano predeterminado.

SREs fazem uso de tais *runbooks* quando recebem um *page* (notificação de um ou mais alertas) e, ao visualizar este documento, rapidamente absorvem informações sobre o serviço para guiar suas próximas decisões.

Um bom *runbook* inclui detalhes como:

- Breve descrição do serviço. O que ele faz? Com quem ele se comunica? Outros serviços dependem dele?
- Onde o serviço está sendo executado? (Caso a organização possua serviços cujos *clusters* existam em múltiplos *data centers*, regiões etc.)
- Quem é responsável por este serviço? Quem são os(as) engenheiros(as) de plantão caso seja necessário obter assistência com a triagem ou para escalar um incidente.
- Links para o repositório do código-fonte do serviço, *pipelines* de *deployments* recentes, painéis de monitoramento do serviço (*dashboards*), logs, *traces*, *continuous profiling* etc.
- Problemas conhecidos. Caso exista alguma anomalia recorrente, incidentes que ocorreram recentemente ou quaisquer detalhes sobre algum problema remanescente que ainda impacta o serviço intermitentemente.
- Alertas associados com o serviço e os respectivos passos que devem ser tomados caso algum destes alarmes dispare.

Tais documentos não precisam, necessariamente, de todos estes detalhes. Alguns *runbooks* podem conter diagramas arquiteturais mostrando gráficos de dependências, ou alguma referência visual

dos componentes lógicos do serviço ou até mesmo algum calendário especial de plantão para feriados caso algum(a) engenheiro(a) específicos(as) forneçam uma cobertura durante datas comemorativas.

Entre todos esses detalhes, é ideal que autores(as) de *runbooks* priorizem os passos associados a cada alerta, já que facilita bastante a missão de evitar incidentes ou mitigá-los rapidamente para restaurar a experiência do cliente. Para ilustrar um exemplo em que tal documentação é bastante valiosa em caso de respostas imediatas, imagine um serviço de *back-end* que expõe uma função para um website ou aplicativo e que esta função possui um interruptor de funcionalidade (*Feature Toggle*), permitindo que a função mude seu comportamento em tempo real (sem a necessidade de um novo *deployment*), então o *runbook* pode descrever algo como:

Alerta	Prioridade	Descrição	Como proceder
SR da Função X < 99%	alta	Esta função experimental impacta funcionalidades A e B	Desligue o *feature toggle* `funcX:enable_new_behavior` e observe o gráfico https://grafana.loja-api.com/servico-backend-dashboard?viewPanel=3 para monitorar a recuperaçãc do serviço. Se continuar enfrentando problemas entre em contato com @fulano ou @cicrano

Obviamente, nem todo alerta pode ser documentado em um *runbook* com os passos exatos que levem à resolução de um problema, mas através da documentação destes passos acionáveis (*actionable steps*), os(as) responsáveis pelos serviços podem raciocinar sobre a sequência de ações investigativas que podem ser tomadas durante a triagem de um determinado alerta e ajudar os(as) leitores(as) de um *runbook* a tomarem os melhores caminhos para desvendar a causa raiz ou tomar decisões em relação a uma mitigação em potencial.

Quando tais passos acionáveis são documentados pelos(as) próprios(as) desenvolvedores(as) de um serviço, são contextualizados com a noção das dependências do serviço, os componentes de infraestrutura e até mesmo possíveis menções de problemas conhecidos com outras entidades do sistema. Antes que um *runbook* seja formalmente publicado ou adotado para um suporte 24/7, é ideal que SREs revisem e forneçam sugestões para melhorar o documento. Outra opção é uma simulação de incidente

(*Fire Drill*), que é uma ótima maneira de testar *runbooks* (será discutido em detalhes no capítulo 13).

10.4 Lidando com alertas durante o plantão (on-call)

SREs geralmente estão na primeira linha de defesa e chegam à "cena do crime" antes de outros(as) engenheiros(as), lidando com o *stress* de potencialmente encontrar eventos disruptivos que podem ocorrer no meio da madrugada. Devem estar preparados(as) para responder a chamadas a qualquer momento, utilizar as ferramentas de observabilidade de forma estratégica e eficiente, invocar algum especialista (do inglês, *Subject Matter Expert*) através do sistema de escalação e, caso a triagem revele um problema severo e um incidente seja declarado, SREs assumem o papel de Comandante de Incidente e devem tomar decisões críticas ao coordenar os passos que devem levar à mitigação do problema. Toda a carga cognitiva envolvida neste processo combinada com a enorme responsabilidade de proteger o negócio da empresa, pode ser uma tarefa bastante difícil para qualquer um, portanto SREs devem conhecer o que fazer e o que não fazer quando se trata de organizar um plantão.

As soluções de gerenciamento de plantão (*on-call*), além de emitirem notificações, são utilizadas também para o gerenciamento de times, cronogramas de plantão e, através de políticas de escalação, pode-se definir quais os diferentes contatos que receberão notificações e em que sequência devem ser contatados de acordo com seu nível. O **PagerDuty**, por exemplo, é uma solução que tem grande adoção na indústria e oferece todas essas funcionalidades: o operador pode definir um serviço, o time responsável pelo serviço, um agendamento (*schedule*) contendo

os(as) engenheiros(as) e um período de tempo em que cada um(a) ficará *on-call*, as políticas de escalação (*escalation policies*) que determinam em que ordem alguém deverá ser notificado e uma infinidade de integrações com os mais variados sistemas para enviar notificações. SREs devem se familiarizar com tais ferramentas para configurar e ajustar qualquer plantão de acordo com suas necessidades.

Um processo de escalação pode acontecer da seguinte forma: digamos que um alerta foi configurado para emitir uma notificação para o time responsável pelo serviço X e a política de escalação associada com este serviço faz com que o sistema tente contatar o engenheiro de plantão primário, caso ele não atenda seu celular ou não confirme recebimento via texto depois de 10 minutos, o engenheiro secundário receberá a chamada e, se passarem mais 10 minutos sem nenhuma resposta, a notificação é escalada para o gerente do time. Isso é um padrão bastante adotado quando um plantão de engenheiros(as) é definido, pois permite que outro membro do plantão possa servir como *backup* caso o primário esteja momentaneamente indisponível e no raro cenário em que ambos não conseguem confirmar o recebimento de chamado, ele é escalado para o gerente, que deve saber como proceder para encontrar alguém com o conhecimento técnico para analisar o alerta e conduzir uma triagem (e discutir com o time como buscar melhorias caso ninguém responda ao alerta).

Quando um calendário de plantão e revezamento é elaborado, é uma boa prática evitar que o(a) mesmo(a) engenheiro(a) fique de plantão por múltiplos turnos ou dias consecutivos pois alguns turnos podem ser bastante agitados. Alguns incidentes podem durar horas e horas de investigação e coordenação de múltiplos times de engenharia, exigindo muita energia de todos os outros envolvidos. SREs que assumem o papel de Comandantes de Incidente, geralmente ficam encarregados de produzir relatórios sub uma certa frequência para compartilhar qualquer progresso no gerenciamento do incidente com o restante da organização, incluindo as camadas

gerenciais, outros departamentos ou, em certos casos, uma comunicação externa com usuários através de uma página de *status* pública.

Uma vez que incidentes duradouros de alta severidade são resolvidos ou que dúzias de alertas são investigados, engenheiros(as) podem chegar próximos da exaustão. Para evitar isso, um bom plantão deve possuir um revezamento apropriado para que o *pager* possa ser passado para outra pessoa e gerentes também devem oferecer algum período de descanso, fornecendo uma chance para que todos possam recarregar suas baterias. Os períodos de cada turno podem variar de acordo com a capacidade de cada organização, porém é importante lembrar dessa ressalva ao planejar um calendário de plantão.

Outra recomendação importante é a definição de engenheiros(as) *on-call* primários e secundários. Definir um plantão com um par de engenheiros permite que o primário possa dividir tarefas durante um turno atribulado ou, em caso de algum problema de alta severidade cuja causa raiz seja obscura, ter alguém para discutir diferentes abordagens para investigar o problema. Esta recomendação também é aplicada para o plantão de times de desenvolvimento e seus próprios níveis de escalação. SREs devem lembrar os responsáveis pelos serviços de que a existência da um(a) único(a) engenheiro(a) *on-call*, sem qualquer revezamento, é uma "receita para o desastre" e que pode, eventualmente, levar aquele(a) engenheiro(a) ao desgaste completo (conhecido em inglês como *burnout*), pois SREs podem precisar deste(a) especialista em várias ocasiões em um curto espaço de tempo dependendo dos alertas que podem surgir para um determinado serviço.

Existem maneiras de evitar tais episódios em que apenas um(a) engenheiro(a) possui o conhecimento técnico para fornecer suporte para um serviço, a liderança dos times responsáveis pelo serviço deve procurar envolver outros(as) engenheiros(as) nas investigações de problemas (*debugging* ou *troubleshooting*) para quebrar os silos de conhecimento, ou apenas se certificar de que algum(a)

engenheiro(a) está olhando por cima do ombro de um(a) colega (mesmo que seja virtualmente). Outras opções envolvem sessões de compartilhamento de conhecimento mostrando diagramas de arquitetura do serviço ou "caminhando" sobre os componentes lógicos da aplicação mostrando os pontos de destaque do código (*code walkthroughs*) ou através de programação em pares (*pair programming*). Existem também oportunidades para aprender mais sobre um serviço através de melhorias em seus aspectos não funcionais como modificar o código para introduzir novas métricas, configurar *logs* ou *traces*, melhorar a *pipeline* do serviço para produzir *releases* mais rápidas ou simplesmente atualizar a documentação técnica sobre o serviço e produzir *runbooks* para ajudar SREs a responderem aos alertas gerados pelo serviço. Essas ações podem ajudar bastante no processo de maturidade de um plantão mais sustentável e escalável.

> Existem estratégias de plantão compartilhados onde ambos SREs e Desenvolvedores podem receber comunicações na primeira iteração da política de escalação. Este método é particularmente interessante pois é bastante eficiente em cenários em alertas são disparados devido a *deployments* que foram executados recentemente, onde o(a) autor(a) da mudança do código pode ser notificado(a) imediatamente e, com todo o contexto da mudança ainda fresco em suas mentes, podem reagir para aplicar uma mitigação ou reverter a mudança (*rollback*) rapidamente e evitar problemas mais severos. SRE também pode considerar automações para notificar autores(as) de tais mudanças e acomodar tais cenários modificando políticas de notificações durante janelas de *deployment*. Várias estratégias podem ser consideradas para melhorar a experiência de todos os times envolvidos.

Ficar de plantão pode ser difícil, mas todo(a) engenheiro(a) deve manter em mente que ficar de plantão não significa que sua vida "acabou". Não devem presumir que não é possível sair com a família

para um jantar ou ir para o cinema com seu(sua) parceiro(a) só porque devem ficar com o laptop por perto o tempo todo. Qualquer membro de um time de engenharia deve se sentir confortável para se comunicar com seus colegas e pedir para que alguém fique com o *pager* por algumas horas e, quando a oportunidade surgir, cobrir alguma parte do turno de outros de forma recíproca. SRE também leva em consideração a qualidade de vida em seus processos, portanto, além de proteger outros membros da engenharia de exaustão (*burnouts*) através de boas estratégias de plantão, SREs devem cuidar de si mesmos e, quando for preciso, mandar uma mensagem para um colega pedindo para que eles deem uma olhadinha no *pager* se precisarem dirigir até a escola para buscar os filhos ou ir para academia etc. Esta preocupação com a qualidade de vida de membros de times de engenharia deve ser compartilhada pela gerência e outras camadas de liderança da organização.

Parte 5: Gerenciamento de incidentes

Capítulo 11
Ferramentas de SRE

SREs são engenheiros(as) de software, portanto o desenvolvimento e a manutenção de ferramentas é uma tarefa recorrente no dia a dia de SREs. Existem várias ferramentas que são implementadas ou adotadas por times que estão dando os primeiros passos com SRE, como catálogo de serviços, sonda de monitoramento sintético, gerador de relatórios de SLOs, gerenciador de incidentes, provisionador de *dashboards* e alertas, interruptores de funcionalidades, ferramentas de *deployment* com mitigação automática de incidentes etc.

Algumas destas ferramentas podem ser desenvolvidas em parceria com outros times, como um time de DevOps ou time de produtividade, por exemplo, já que estão geralmente envolvidos com a otimização da experiência de desenvolvimento, CI/CD e até mesmo Observabilidade. Como visto anteriormente, DevOps e SRE possuem um escopo bastante comum, então engenheiros(as) DevOps são grandes aliados(as) nesses projetos.

Os exemplos mencionados anteriormente são bastante utilizados na indústria, mas as ferramentas desenvolvidas por SREs devem ser concebidas de acordo com as necessidades da organização. Basicamente, qualquer solução que melhore processos internos para facilitar atividades operacionais ou automações que eliminem tarefas repetitivas, tornando-as mais eficientes e menos suscetíveis a erros humanos, torna-se elegível para complementar o arsenal de instrumentos que estão à disposição da Engenharia de Confiabilidade de Sites.

Para orientar o *design* e implementação de tais ferramentas, é importante entender as atividades do dia a dia envolvendo a detecção, gerenciamento e resolução de incidentes, assim como o suporte fornecido para serviços e as respectivas entidades envolvidas nestes processos. Para ilustrar, vamos examinar um cenário onde um SLO é violado e uma sequência de ações são tomadas:

- Vamos supor que um alarme associado ao serviço X é disparado, então SRE recorre à telemetria (*Dashboard*) e a documentação que fornece detalhes sobre o serviço e os passos acionáveis de seus alertas (*runbook*) para realizar uma triagem e definir a severidade do incidente.
- O serviço X está categorizado como *tier* 1, porém as métricas mostram que apenas um pequeno percentual de usuários foi afetado pelo problema, o que significa que, apesar do impacto substancial, o sistema está severamente comprometido, mas não apresenta indisponibilidade total. Portanto, um incidente de severidade 2 é criado para lidar com o problema.
- Depois de explorar algumas opções, SRE que agora atua como Comandante do Incidente, conclui que não foi possível encontrar uma mitigação que pode ser aplicada facilmente, então decide escalar a chamada e trazer especialistas (responsáveis pelo serviço, ou *Subject Matter Experts*) para ajudar com o incidente. Neste cenário, especialistas que também estão de plantão (*on-call*) recebem notificações (*Page*) em seus dispositivos móveis e juntam-se à pessoa que atua como Comandante do Incidente para auxiliar na investigação.
- Após trabalharem juntos por alguns minutos, encontram uma opção para restaurar a experiência do usuário. A mitigação é aplicada e a telemetria é observada para confirmar que o serviço X foi, de fato, restaurado para um estado em que sua performance está de acordo com os parâmetros de seus SLIs e SLO.
- Em outro momento, quando o incidente já não está mais em progresso, os times envolvidos decidem que ele está elegível

para uma reunião de *Post Mortem*, onde desenvolvedores, o Comandante do Incidente e gerentes podem revisar os eventos que levaram até o surgimento do incidente, o impacto e as ações que foram tomadas para sua resolução. A reunião é organizada por quem atua como Comandante do Incidente, a linha do tempo de eventos é discutida e atividades de melhoria contínua são identificadas e atribuídas aos respectivos membros da engenharia.
- O incidente é fechado logo depois que as tarefas de melhoria são concluídas.

Com base neste cenário, podemos concluir então que existe uma série de entidades nesta jornada: mecanismos de deteção de violação de SLO (alertas), o serviço e seus metadados (como *tier*, por exemplo), um incidente que transiciona entre diferentes estados ("aberto", "mitigado" e "fechado"), documentação técnica (*runbook*), telemetria utilizada para investigações e a inspeção do estado do serviço (métricas, *traces*, *logs* etc.), contato para escalação (ferramenta para enviar *pages* e gerenciamento de calendário de plantão), reunião de *Post Mortem* etc. Tais entidades e suas interações nestes fluxos de processos devem ser mapeadas, assim como seus respectivos artefatos que compõem a Engenharia de Software da organização. Desta forma, podemos identificar oportunidades para desenvolver, ou avaliar e adquirir ferramentas existentes, que otimizem os processos de gerenciamento de incidentes. Isso é um passo natural para times que possuem uma visão macro, cujo escopo abrange toda a organização, e que precisam estar cientes de todas as funcionalidades dos produtos que são críticos para o negócio para que possam fornecer suporte e manter sistemas operando como esperado. Para explorarmos mais este mapeamento de entidades e o gerenciamento de serviços, vamos falar um pouco mais sobre "metaengenharia".

11.1 Metaengenharia

Partindo novamente da premissa de que SRE se comporta como um time de desenvolvimento de software que tem como intenção automatizar as responsabilidades operacionais para fins de manutenção e suporte de serviços, um(a) engenheiro(a) de confiabilidade de sites observa o serviço como uma entidade que se correlaciona com outros componentes da Engenharia de Software. Para visualizarmos estas entidades, podemos considerar uma hierarquia onde a organização é composta por times que são responsáveis por um ou mais serviços e, para cada serviço, pode-se definir uma série de atributos, por exemplo: a infraestrutura do serviço (sua plataforma de *runtime* e a pilha de tecnologia, ou, do inglês, *Tech Stack*), suas métricas, logs, *traces* (encontrados em *dashboards*), documentação (*runbooks*), seus canais de comunicação e escalação (*on-call*) ou níveis de prioridade (*tier*). Esses componentes podem ser visualizados como abstrações para que engenheiros(as) de SRE possam criar o design de um sistema.

Dependendo do mapeamento de componentes e das intenções de automação, pode existir a necessidade de associar mais atributos à entidade "Serviço", como: linguagem de programação, repositório de código, canal de mensageria, ou qualquer outra informação que possa enriquecer o conjunto de metadados associados a um serviço. A definição desses metadados e o mapeamento deles para cada serviço, ou algum outro componente da engenharia, serve como uma referência para raciocinar sobre os processos que podem ser automatizados e que ferramentas podem ser construídas para facilitar atividades rotineiras. Por isso, SREs são aqueles(as) que trabalham com a "engenharia da engenharia de software", pois observam todas as partes que atuam no processo de desenvolvimento de software e operações e encontram oportunidades para organizar e interagir programaticamente com esses componentes.

Aqui está um exemplo mais concreto de tais designs e ferramentas: SREs do Google desenvolveram uma ferramenta interna chamada "Outalator" para catalogar *outages* (eventos de indisponibilidade) e ajudar na tarefa de gerenciamento de incidentes. Mais detalhes podem ser encontrados no capítulo 16 do livro de SRE: *Tracking Outages* (Murphy, 2016). A ferramenta Outalator captura notificações de todos os alarmes e ajuda SREs com a agregação, rotulamento (*tagging*) e análise de incidentes, ajudando o time de SRE e todo o resto da organização com consultas para fins de obter *insights* sobre incidentes: identificar serviços impactados por um incidente, agrupar alertas em um único incidente, relatórios, correlações e definição de diferentes categorias de causa raiz etc.

Se você deseja iniciar sua jornada de desenvolvimento de ferramentas de SRE, mas não sabe por onde começar, comece a mapear as entidades e artefatos de engenharia da sua organização (os responsáveis por um conjunto de funcionalidades, assim como os serviços e seus atributos, dependências de infraestrutura etc.). No decorrer deste capítulo, nós vamos apresentar mais exemplos de ferramentas, discutir seus benefícios e explorar cenários em que automações podem ser implementadas para resolver problemas encontrados por SREs, porém lembre-se que adotar uma ferramenta existente também é uma opção.

11.2 Sobre construir as próprias ferramentas ou adotar soluções existentes no mercado

Não precisa reinventar a roda se uma solução existente já atende às necessidades da sua organização. Entretanto, fique ciente de que qualquer solução de terceiros sempre terá suas limitações. Por exemplo, a solução contratada pode não atender a um caso de uso específico de algum processo ou, em caso de alguma indisponibilidade (lembre-se que todos os serviços, incluindo aqueles

de terceiros, estão sujeitos a eventuais incidentes), a organização pode enfrentar um impacto de produtividade ao interromper suas atividades ou consumir mais tempo para concluir operações contornando a ausência de uma ferramenta. Se um time de SRE deseja que a ferramenta ofereça uma determinada funcionalidade que não é suportada pela solução, não poderá operar como desejado, tornando-se refém das diretrizes estabelecidas pela ferramenta de terceiros.

Algumas ferramentas de terceiros podem expor pontos de extensibilidade que permitem que os próprios clientes conectem customizações, outras são disponibilizadas em projetos de código aberto, possibilitando uma bifurcação (*fork*) da lógica original que pode ser alterada para necessidades especificas ou, se você deseja contribuir para o projeto, pode submeter uma mudança para que seja revisada pelos mantenedores e, caso seja aceita, pode ser incluída na próxima *release* da ferramenta. Algumas organizações que oferecem ferramentas proprietárias aceitam pedidos de novas funcionalidades (*feature requests*) e, dependendo da empresa, podem receber tais pedidos através de gerentes técnicos (*Technical Account Manager*) que ficam à disposição para facilitar qualquer comunicação com o time de sua engenharia. Esses são alguns cenários em que as limitações de ferramentas de terceiros podem ser amenizadas ou contornadas, mas em casos bem específicos em que alguma integração com processos ou dependências internas se torna necessária, o time deve considerar a construção de uma nova ferramenta que atenda aos seus propósitos.

Quando um time de SRE desenvolve a própria ferramenta, ele possui mais autonomia sobre a flexibilidade de seu design, suas funcionalidades e como a ferramenta poderá ser adaptada para as necessidades específicas de sua engenharia. Porém, isso vai exigir mais esforço do time de SRE para desenvolver a ferramenta, assim como fazer a manutenção e o custo de infraestrutura. A decisão deve ser feita de acordo com o equilíbrio que faz mais sentido para sua organização.

Nada substitui a autonomia de trabalhar com suas próprias ferramentas. É por isso que *Big Techs* com práticas de SRE bastante amadurecidas como Google, Meta ou X (historicamente conhecidas como *Facebook* e *Twitter*, respectivamente), notoriamente desenvolvem suas próprias ferramentas, desde sistemas de Observabilidade até utilitários para gerenciamento de incidentes, utilizando poucas ferramentas de terceiros para tarefas bem específicas que, em sua maioria, não são tão críticas para suas próprias práticas de SRE. Tais decisões são baseadas em quanto vai custar para a organização manter um contrato com um SaaS (*Software as a Service*), integrar o produto com suas necessidades internas e manter essa integração, em comparação com o custo de desenvolver e manter ferramentas desenvolvidas internamente.

11.3 Sugestões de ferramentas

As seções a seguir contêm dois exemplos de ferramentas que podem ser desenvolvidas para facilitar tarefas de SRE. Não entraremos em detalhes técnicos sobre como exatamente as ferramentas devem ser desenvolvidas, vamos apenas discutir a ideia por trás delas e quais os problemas que são amenizados quando consideramos tais ferramentas para obter mais processos automatizados.

O Catálogo de Serviços e como orquestrar operações

SRE tem como um de seus focos o gerenciamento de incidentes que são causados por eventos inesperados associados com os serviços, o que faz com que a entidade "Serviço" torne-se uma peça fundamental nesta equação. Portanto, assim como em um inventário, o catálogo de serviços é um ótimo ponto de partida para gerenciar os componentes de um sistema, registrar quem são seus responsáveis e mapear artefatos associados com eles.

Esses catálogos são bastante úteis para mapear metadados e fazer com que os artefatos de cada serviço sejam facilmente localizáveis, melhorando sua "Encontrabilidade" (*Discoverability*), ou seja, eliminando a necessidade de investigar que desenvolvedores(as) são responsáveis pelo serviço e evitando o atrito para encontrar *dashboards*, *runbooks*, *tiers*.

Durante uma triagem ou incidente, não podemos perder tempo perguntando quem é dono de qual serviço ou onde está determinado artefato, pois qualquer sobrecarga de comunicação prejudica o MTTR (*Mean Time To Resolution*). O catálogo de serviços pode ser consultado para que tais informações sejam obtidas rapidamente, para auxiliar SREs e outros membros da engenharia a conduzirem alguma investigação ou, em um cenário mais orientado à automação, o catálogo pode ser instrumentado programaticamente para orquestrar operações, possibilitando a execução de tarefas através de ferramentas que se comunicam entre si.

O diagrama a seguir ilustra como esses metadados podem ser apresentados em termos de entidades e seus respectivos atributos, similar aos clássicos diagramas de UML (*Unified Modeling Language*) comumente utilizados em designs para Programação Orientada a Objetos.

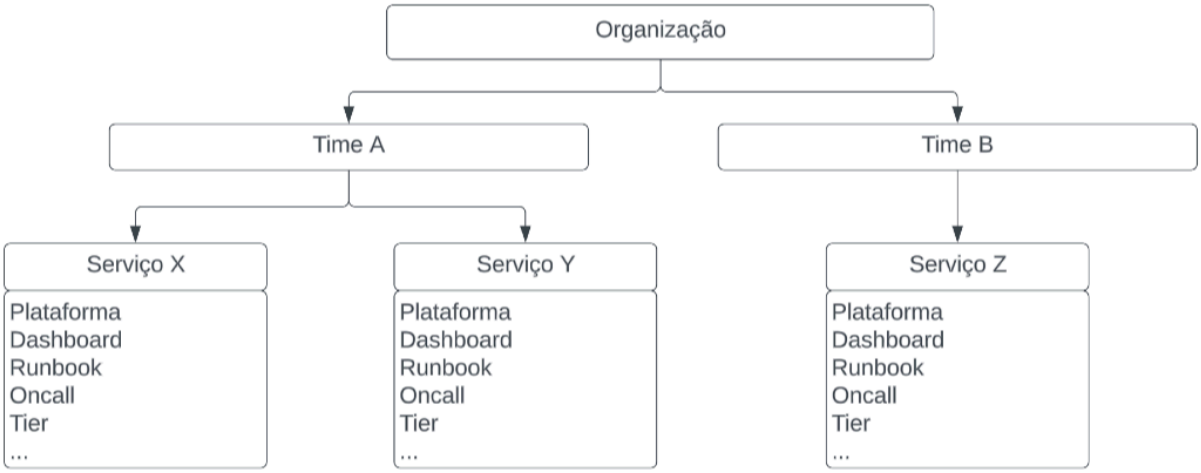

Figura 11.1: Ilustrando serviços e seus respectivos metadados

Uma solução bastante famosa no mercado é o *PagerDuty*, que possui um Diretório de Serviços (https://support.pagerduty.com/main/docs/service-directory) e permite que o usuário defina uma política de escalação (*escalation policy*) para cada serviço, assim como um time responsável pelo serviço e um calendário de plantão para seus membros. Serviços catalogados no *PagerDuty* possuem "integrações" que podem enviar e receber requisições para inúmeras ferramentas de terceiros, como receber uma requisição de um ponto de contato do Grafana para iniciar um incidente ou enviar mensagens para um sistema de mensageria como o Slack.

SREs devem procurar oportunidades para orquestrar suas automações, ou seja, conectar as funcionalidades de suas ferramentas para otimizar processos. Vamos considerar um exemplo em que ferramentas de SREs podem ser utilizadas para diminuir o tempo de triagem e mitigação de um incidente. Imagine o seguinte cenário:

- Serviço X manifesta um problema. Suas métricas são avaliadas por um alarme que, por sua vez, ativa notificações enviando um *page* para o(a) engenheiro(a) que está de plantão e coloca uma mensagem em uma fila de eventos.

- Uma ferramenta consome a mensagem da fila e consulta o catálogo de serviços, obtém dados sobre o serviço, o time responsável pelo serviço (*owner*), seus pontos de escalação (*on-call*), seu repositório de código, o canal de comunicação com o time responsável, o endereço do *runbook* e o *dashboard* do serviço.
- A mesma ferramenta pode enviar esses dados para o(a) SRE e economizar tempo de triagem, ou até mesmo verificar *deployments* recentes e reverter a mudança de código automaticamente, restaurando a experiência dos clientes do serviço X rapidamente.
- Outros serviços podem ser impactados pelos problemas de performance do serviço X, mais automações podem verificar as métricas de outros serviços e produzir um relatório do impacto do negócio que será utilizado na reunião de *Post Mortem*.

Orquestrar ferramentas é uma tarefa fascinante, mas que se torna desafiadora quando existem muitas permutações de elementos de metaengenharia (diferentes times utilizando tecnologias com diferentes abordagens de CI/CD), portanto nem toda solução existente no mercado poderá atender às necessidades específicas de sua organização. É ideal que exista uma tentativa de evangelizar mais padronização entre diferentes times de engenharia, pois isso vai diminuir significativamente os esforços de design, implementação e manutenção de todos os pontos de interação que tais ferramentas devem acomodar para suportar todos os serviços de uma organização.

Para orientar o seu design, tente entender as dinâmicas de sua organização. Por exemplo: como estão definidas as relações entre os times de desenvolvimento de software e os serviços que estão operando em produção? Existem vários modelos para definir que time é responsável por algum serviço, mapeando a posse ou apropriação (do inglês, *ownership*) deles. Para citarmos alguns:

1. Um time para cada serviço: padrão encontrado em empresas grandes com mais recursos.

2. Um time é dono de múltiplos serviços: padrão mais comum. Times podem ser definidos de acordo com o escopo dos serviços ou a natureza dos mesmos, por exemplo, o time de *front-end* é responsável pelo website da empresa e um portal de clientes, enquanto o time de *back-end* é o dono dos serviços que acessam bancos de dados, interagem com sistemas de terceiros etc. Outro time que pode existir é o time de Análise de Dados, cujos membros devem gerenciar os aplicativos relacionados a *data pipelines* que também estão sujeitos a eventos que podem gerar incidentes.
3. Serviços órfãos: algumas organizações podem operar com "sistemas legados" ou de manutenção esporádica. Os engenheiros responsáveis por esses serviços, que detêm algum conhecimento histórico de seu código (*Subject Matter Experts*), podem não estar mais disponíveis (alocados em outros projetos ou não trabalham mais na empresa). Tais serviços oferecem um atrito de comunicação, impactam a evolução da maturidade de processos e tornam-se um risco imediato para SRE, especialmente se os serviços pertencerem a *tier* 1 ou 2. SREs devem colaborar com a camada de gerência para encontrar uma estratégia de plantão para fornecer suporte a tais serviços.

Dependendo do propósito da ferramenta, entender as relações de responsabilidade (*ownership*) dos serviços é fundamental para seu design e desenvolvimento. Para atingir resultados como o do cenário fictício anterior, o catálogo deve possuir os atributos relacionados a cada serviço, como ilustrado anteriormente no diagrama. Por isso, a colaboração com os donos dos serviços é vital, pois um catálogo de serviços só será totalmente útil se for populado e atualizado corretamente, então um esforço conjunto ajuda bastante já que a introdução de tal catálogo deve pavimentar o caminho para o gerenciamento de incidentes e orquestração de ferramentas. Entretanto, não é necessariamente crítico que todo serviço seja registrado no catálogo, já que as prioridades devem ser avaliadas de acordo com as funcionalidades mais críticas do produto como um

todo, baseadas nas jornadas de usuários e nos requisitos do negócio.

Revisor de requisitos para operar em Produção

Como mencionado no capítulo 2 (*Competências de SRE*), uma das tarefas recorrentes é a avaliação dos pré-requisitos para certificar um serviço como "Pronto para Produção" (termo conhecido em inglês como *Production Readiness*). Esta certificação é basicamente o processo em que SREs fornecem um selo de aprovação para informar outros membros da engenharia que um determinado serviço cumpre os requisitos esperados para operar no ambiente de Produção, onde pode cumprir todos os requisitos perfeitamente ou atender ao critério de forma satisfatória.

Este processo não apenas oferece uma segurança psicológica de que determinados artefatos estarão disponíveis para auxiliar na investigação e resolução de problemas, mas, ao trabalhar com esta lista de verificações, os responsáveis pelos serviços podem também *prevenir* incidentes puramente através do exercício de configurar serviços de uma maneira mais alinhada com os requisitos operacionais de Produção.

O ato de realizar tal certificação de serviços geralmente envolve uma série de passos, que podem ser descritos através de perguntas:

- O serviço possui monitoramento? Para avaliar se o serviço está operando de forma satisfatório, é necessário que alguma telemetria seja emitida ou que alguma sondagem possa ser executada em suas funcionalidades. Isso inclui a presença de métricas, *traces*, *logs*, um *dashboard* e até mesmo interfaces para acompanhar *deployments* e outros eventos significativos.
- Existe alguma documentação técnica? Caso outros(as) engenheiros(as) precisem obter detalhes básicos sobre o serviço (como ele funciona, quais são suas dependências, sua arquitetura etc.) ou, em caso de eventos inesperados ou alertas, se precisarem de passos documentados para realizar uma

triagem ou resolver problemas, um *runbook* é um recurso fundamental para orientar SRE e outros membros de outros times.
- Os responsáveis por este serviço possuem uma estratégia de plantão? Em caso de incidentes, em que problemas precisam ser escalados e torna-se necessário o envolvimento de especialistas para fins de investigação e a busca por uma mitigação rápida, os membros do time que possui conhecimento para resolver problemas associados com o serviço devem carregar consigo um *pager* para que possam ser chamados em caso de emergências.
- Foi feita alguma revisão dos requisitos de infraestrutura para este serviço? Alguma verificação deve ser conduzida para atestar que o serviço possui recursos computacionais suficientes para lidar com a carga de trabalho de Produção, entre outras verificações relacionadas a requisitos não funcionais como escalabilidade, disponibilidade, segurança etc.
- O processo de *release* deste serviço está alinhado com boas práticas? Problemas podem ser capturados mais cedo no ciclo de vida de desenvolvimento, portanto ele deve possuir testes unitários, uma *pipeline* de integração contínua, deve ser testado em ambientes de Pré-produção e seus *deployments* devem ser automatizados para facilitar a entrega de mudanças de código para eliminar *bugs* e corrigir quaisquer falhas que impactem clientes em Produção.

Estes são apenas exemplos de algumas avaliações que podem ser críticas dependendo da organização e, como podemos ver, cada um destes tópicos envolve vários artefatos que devem ser verificados, o que implica em um esforço bastante manual, sem mencionar o fato de que este é um exemplo de certificação de um único serviço e que organizações de engenharia de software trabalham com dezenas, centenas ou, potencialmente, até mesmo milhares de serviços. Então um processo como esse geralmente envolve uma grande permutação de serviços e a avaliação de requisitos que podem possuir seus próprios níveis de granularidade como, por exemplo,

um *runbook* que contém alertas, passos acionáveis, *dashboards*, pontos de contato para escalação etc.

Em situações como essa, geralmente existem duas opções: desenvolver ferramentas que configurem os artefatos de um serviço durante a sua criação para que já sejam certificados no momento que ganham vida, ou criar um sistema que deve automatizar cada uma destas verificações, para que SREs possam eliminar o esforço repetitivo da certificação de múltiplos serviços. A primeira opção é bastante apropriada em casos em que a organização não possui muitos serviços e a engenharia está em seus estágios iniciais de amadurecimento de processos. Esse cenário é conhecido em inglês como *green fielding* pois sugere um campo verdejante onde tudo ainda vai florescer. Por outro lado, em casos em que já existe um número significativo de serviços com seus respectivos artefatos já estabelecidos, a segunda opção torna-se mais viável pois as automações podem inspecionar os artefatos e fornecer um relatório sem a necessidade de recriar todos os serviços de forma disruptiva.

Um sistema revisor de requisitos pode utilizar APIs de outras ferramentas para obter os dados necessários, programaticamente executar asserções e determinar se o artefato do serviço está configurado de acordo com o que está oficialmente documentado como um pré-requisito para que o serviço possa operar em Produção. A ferramenta pode emitir métricas com a pontuação de cada requisito e o resultado total para cada serviço, permitindo que gráficos (ou alertas) sejam criados com tais pontos, assim como relatórios que devem ser enviados para algum canal do sistema de mensageria da organização. Esse relatório pode ser produzido sob demanda ou dentro de algum agendamento para que os responsáveis pelos serviços observem como o serviço está avaliado em relação aos requisitos estabelecidos pela organização.

Aqui está um diagrama que pode ser usado como inspiração para o design e implementação de tal ferramenta.

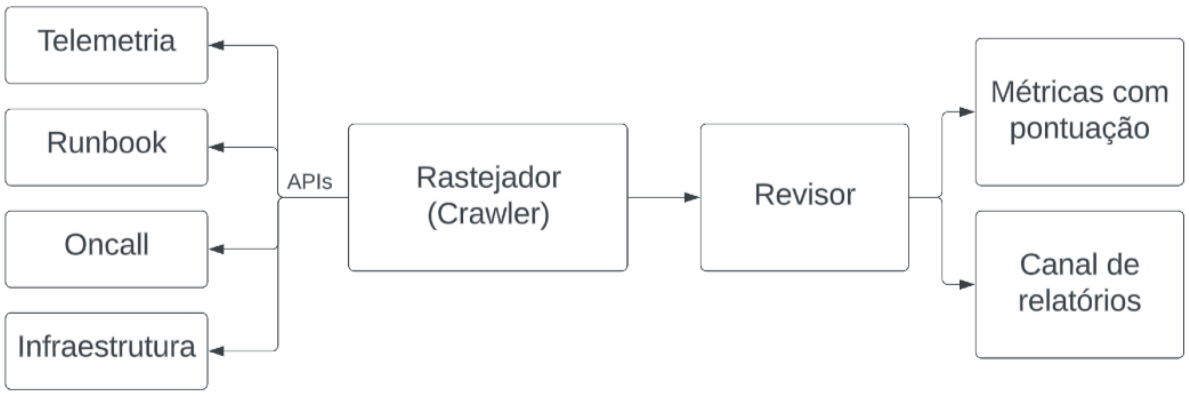

Figura 11.2: Exemplo de como um revisor de requisitos de serviços pode funcionar

Os relatórios podem ser formatados para exibir uma pontuação, como uma "nota" para o serviço e seus respectivos requisitos de acordo com a qualidade de suas configurações. Para ilustrar, imagine que um serviço possui métricas, mas não gera nenhum log ou que seu *runbook* documenta apenas uma parcela dos alertas e apresenta total ausência de um diagrama de arquitetura. Nessas situações, um serviço não deve obter a pontuação máxima. A ferramenta revisora pode conceder pontos baseados em "pesos" dependendo de quão crítico um determinado requisito pode ser considerado em casos de triagem e gerenciamento de incidentes. Vamos supor que o monitoramento sintético pode ser um critério estabelecido com peso `10`, enquanto o *runbook* pode ter peso `7` e o cálculo da pontuação do serviço será influenciado por esses modificadores. Esses são apenas exemplos de como a revisão de requisitos pode ser mais sofisticada, porém este nível de granularidade é algo que pode ser implementado em outra fase de amadurecimento da ferramenta.

Esses requisitos devem ser revisitados e atualizados colaborativamente, SREs devem convidar desenvolvedores(as) para discutir os critérios de excelência operacional e implementar novas verificações na ferramenta. Desta forma, SRE pode reportar os serviços de acordo com suas respectivas pontuações, categorizando aqueles que não cumprem requisitos críticos ou que precisam de

ajustes nas configurações de artefatos, de maneira que os resultados possam ser facilmente interpretados por todos os membros da organização, não apenas para os(as) desenvolvedores(as) responsáveis pelos serviços, mas até mesmo para gerentes que devem observar tais relatórios e tomar decisões estratégicas sobre como pretendem alocar esforços no próximo ciclo de desenvolvimento.

Algumas organizações podem tomar decisões mais rígidas em relação à avaliação de uma certificação de *Production Readiness*. O quadro de resultados nesses relatórios pode ser usado como critério para postergar alguma *release* caso o serviço não obtenha uma pontuação acima de algum limiar (exemplo: 80% ou 90% da pontuação máxima). A implementação de práticas que bloqueiam *deployments* dependem da cultura da organização: tais sugestões podem antagonizar desenvolvedores(as) e criar outros problemas caso não exista um consenso sobre cada um dos requisitos operacionais ou em caso de falsos positivos gerados por algum *bug* na automação do revisor. SRE pode utilizar tal ferramenta apenas para fins de aconselhamento e considerar uma abordagem mais sensível onde nem todos os requisitos e suas respectivas notas devem necessariamente criar algum atrito na jornada de desenvolvimento.

SREs devem estar prontos(as) para navegar discussões que podem ser disruptivas dependendo de como os times de engenharia estão acostumados a operar, pois nem toda organização vai adotar cada uma das práticas de SRE facilmente através de uma imposição de figuras de autoridade como um(a) CTO (*Chief Technology Officer*) ou algum(a) diretor(a) de desenvolvimento. Em alguns casos, SREs devem encontrar aliados para evangelizar a adoção de suas ferramentas e processos, portanto é uma boa prática não construir tais ferramentas "no vácuo" ou, em outras palavras, sem envolver outros times que fazem parte do ciclo de vida da engenharia de software, pois o foco deve estar na colaboração, principalmente com

os times de desenvolvimento (os responsáveis pelos serviços). Criar protótipos ou MVPs (*Minimum Viable Product*) e apresentar a proposta com uma demonstração são métodos bastante comuns para obter *feedback* e ajustar funcionalidades da ferramenta em seus estágios iniciais de desenvolvimento. Tais conversas devem guiar os "contratos sociais" entre os times e amadurecer o ambiente sociotecnológico para facilitar a implementação de SRE na organização.

Capítulo 12
O ciclo de vida de um incidente

Uma das principais responsabilidades de Engenheiros(as) de Confiabilidade de sites é o gerenciamento de incidentes, seja atuando como Comandantes de Incidentes ou educando outros membros da organização sobre como agir durante tais eventos, estabelecendo um processo mais amadurecido e previsível para evitar sobrecarga de comunicação e atritos na hora de lidar com situações em que a experiência de clientes precisa ser restaurada o mais rápido possível. SREs devem estar habituados com a sequência de eventos que compreendem o ciclo de vida de um incidente para gerenciá-lo ou orientar outros membros da organização que possam assumir o mesmo papel.

O ciclo de vida de um incidente é dividido pelos seguintes estágios:

1. Problema reportado
2. Triagem
3. Diagnóstico
4. Tratamento
5. Revisão
6. Conclusão

As atividades que compreendem cada um dos itens enumerados acima são organizadas com o intuito de trazer mais eficiência para o processo de gerenciamento de incidentes, com o foco na redução do tempo de violação de acordos e a prevenção de futuros incidentes da mesma natureza. Uma vez que um problema é reportado, a triagem deve ser conduzida para determinar se ele é elegível a se tornar um incidente, então um diagnóstico é realizado para determinar a causa do problema ou um tratamento aceitável que pode levar a uma mitigação. Quando algum tratamento é aplicado, outro diagnóstico torna-se necessário para verificar se o problema foi, de fato, mitigado ou resolvido, este ciclo de tratamento e

diagnóstico pode se repetir até que o sistema seja restaurado. Quando um incidente é resolvido, uma revisão é conduzida para determinar tarefas de melhoria contínua e prevenção de futuros incidentes e, quando tais tarefas são concluídas, finalmente chegamos à conclusão e ao fim do ciclo. O diagrama a seguir ilustra a sequência de cada um destes passos:

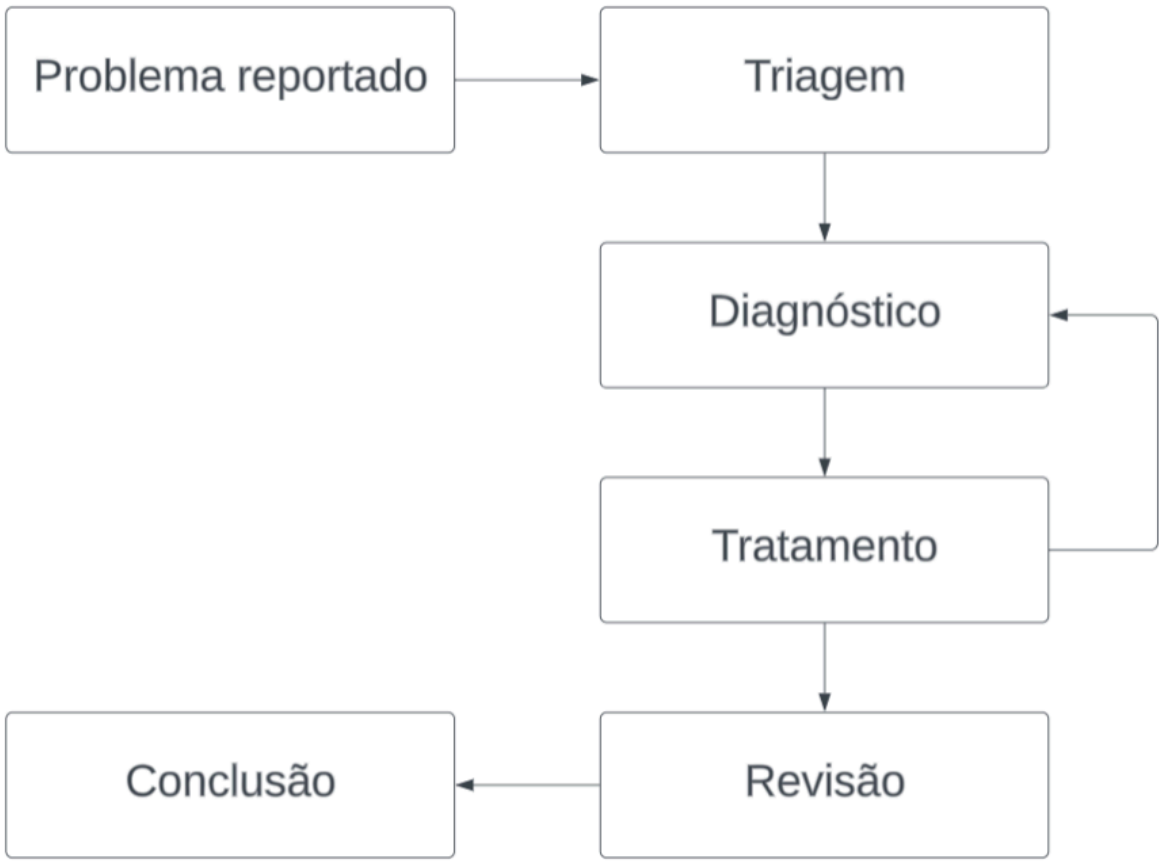

Figura 12.1: Ciclo de vida de um incidente

SREs devem documentar e formalizar este processo para tornar essas orientações transparentes para todos os times da organização que podem, eventualmente, estar envolvidos em um incidente. Nas seções a seguir vamos explorar cada um destes estágios, discutir as responsabilidades do Comandante de Incidente e mencionar alguns exemplos de situações da vida real.

12.1 Problema reportado

Um incidente pode ser originado de algum alerta, cuja notificação é gerada de forma automática ou através de interações humanas. Ambas podem refletir um problema autêntico com alguma funcionalidade de um ou mais serviços (aplicativo móvel, website, API etc.) ou podem ser falsos positivos, esta distinção é determinada no próximo estágio (triagem). Um alerta pode enviar um *page* para engenheiros(as) que estão de plantão (*on-call*) ou ativar outras automações que podem alterar configurações de infraestrutura, enviar mensagens para canais de comunicação ou até mesmo redirecionar o tráfego para outros *data centers* e tentar evitar um impacto mais severo, tais eventos devem ser facilmente observados por SRE para fins de detecção de problemas pois nem todo mecanismo de mitigação automática pode acomodar todas as possíveis situações que podem impactar usuários.

Problemas podem ser reportados por usuários, através de *tickets* de suporte, canais de comunicação com a gerência ou até mesmo em portais de reputação online que registram reclamações de clientes ou redes sociais, porém essas alternativas devem apenas suplementar outras evidências como a telemetria geradas pelos serviços, que deve ser configurada para servir como um método de detecção mais eficaz e confiável. Problemas também podem ser reportados internamente, por exemplo, através do sistema de mensageria da organização, quando alguém descreve algum comportamento anômalo do produto em algum canal de comunicação, mensagens diretas enviadas a engenheiros(as) de plantão ou até mesmo uma chamada telefônica.

Quando a detecção de problemas é realizada por humanos, existe uma chance de que o problema pode estar se manifestando por um longo período de tempo e o seu impacto ainda não foi percebido por uma grande quantidade de usuários, levando a situações possivelmente catastróficas como o processamento de dados corrompidos, precisando de um esforço substancial para correções

retroativas e o envolvimento de mais especialistas, potencialmente em algum momento fora do horário normal de trabalho. Isso nos leva a reiterar a importância da telemetria e alertas como uma maneira mais rápida e pragmática para capturar possíveis incidentes. Em ambos os casos em que o problema é reportado, seja externamente ou internamente, não se deve assumir que o problema seja necessariamente elegível a se tornar um incidente sem uma triagem.

SREs também podem encontrar situações em que vários alertas ou múltiplos contatos enviam muitas mensagens relatando problemas, em tais cenários é importante notificar todos os membros da organização em alguma interface de comunicação comum, acordada e conhecida por todos os times que podem estar envolvidos em incidentes, para deixá-los cientes de que o problema está sendo investigado e evitar o consumo de energia e tempo respondendo a cada uma das mensagens. Lembre-se também que engenheiros(as) de plantão primários podem colaborar com secundários para aliviar a carga de trabalho e "dividir e conquistar" para averiguar alertas ou mensagens relatando problemas, reunindo e organizando informações para proceder com a triagem.

12.2 Triagem

Como discutido no capítulo 2, um incidente é um evento não planejado que afeta a funcionalidade de um serviço e que requer atenção *imediata* e precisa de uma resposta *coordenada*, ou seja, alguns eventos de degradação de performance ou até mesmo indisponibilidade temporária não oferecem risco imediato ao negócio já que existem diferentes tipos de serviços com seus respectivos SLOs e níveis de prioridade (*tiers*).

Quando um problema é reportado por um alerta, engenheiro(as) de plantão devem se perguntar se a situação realmente exige atenção

imediata, verificando a telemetria disponível ou até mesmo a configuração do alerta. Se as métricas indicam que não há uma violação de SLO e não existem outras evidências de que algo está errado, talvez o evento tenha sido parte de algum fenômeno efêmero, ou seja, algo estranho que ocorreu em um curto espaço de tempo mas logo depois o sistema voltou a operar como esperado. As mesmas considerações devem ser aplicadas quando algo é reportado por humanos, pois nós também podemos reportar problemas que não são suficientemente severos e também é uma boa prática, quando possível, tentar reproduzir o problema, já que usuários podem enfrentar interferências específicas de seu ambiente de operação como problemas de conexão ou incompatibilidade de dispositivos etc.

Considerando exemplos mais concretos, vamos supor que um alerta é disparado devido à utilização de 90% de CPU em um cluster de um serviço X. Nessas circunstâncias, se os usuários estão interagindo com o sistema sem encontrar erros ou problemas de performance, então a saturação de recursos de um ou mais servidores não representa algo crítico que justifica a criação de um incidente pois, como visto anteriormente no capítulo 10, este tipo de métrica é considerada um indicador intermediário, ou seja, suas medições não ajudam a correlacionar problemas com a experiência de usuários de maneira eficiente. Agora vamos imaginar um cenário em que um usuário reporta um problema com a interface Web de um serviço Y e que, logo após uma investigação inicial, determina-se que ainda é possível interagir com o sistema sem um atrito substancial e que o problema é puramente cosmético. As entidades envolvidas chegam à conclusão de que uma investigação mais criteriosa pode esperar até que o time responsável esteja disponível em uma hora de trabalho apropriada, e, então, o caso não é elegível para se tornar um incidente já que não exige uma resposta imediata e coordenada.

Outros casos em que problemas reportados não geram incidentes incluem a ocorrência de "falsos positivos". Tais casos de alertas

falsos tornam-se bastante contra produtivos e, quando possível, devem ser eliminados com uma análise das circunstâncias que levaram o alerta à condição de disparo ou ajustes na configuração do alerta. É bastante importante que alertas que geram falsos positivos sejam revisados, pois contribuem para a fadiga por alarmes. Quando alarmes falsos são produzidos por humanos, devem ser educados sobre o problema, explicando como o evento não tem o impacto que foi presumido ou redirecioná-los para alguma documentação. Se tais orientações não estão disponíveis para compartilhamento, é uma boa ideia criar tal documentação para disseminar esse aprendizado para evitar cenários recorrentes que causam interrupções desnecessárias para engenheiros(as) de plantão.

Em organizações com ciclos de desenvolvimento bastante dinâmicos, que criam novos serviços constantemente, SREs nem sempre podem acompanhar todas as evoluções do sistema e se atualizar com os detalhes técnicos de determinados serviço ou jornadas de usuários. Quando engenheiros(as) de plantão não estão familiarizados(as) com as últimas adições na arquitetura e mudanças no produto, enfrenta-se um atrito na triagem de alertas que exigem atenção urgente devido a tais lacunas de conhecimento, pois, durante a análise de algum incidente em potencial, não se espera que muito tempo seja alocado para um estudo denso de funcionalidades e componentes do sistema. Em tais casos, torna-se necessário o envolvimento de especialistas para ajudar na avaliação do impacto na experiência dos usuários. Com esta ajuda, obtém-se a orientação necessária para determinar se é realmente preciso criar um incidente e qual deve ser sua severidade.

A definição de severidade é um processo que divide opiniões e que pode variar entre diferentes organizações, mas a tabela a seguir contém algumas orientações básicas que podem ser usadas como referência para formalizar o procedimento entre os times envolvidos em incidentes.

Severidade	Sev0
Sev0	Indisponibilidade total ou parcial impossibilita que os serviços operem. Exemplo: A infraestrutura que abriga os serviços. Esta categoria também deve incluir plataformas e ferramentas de observabilidade pois como podemos saber se o sistema está operando como esperado sem a habilidade de observar serviços? Por exemplo, se as métricas não estão fluindo, não podemos monitorar uma taxa de sucesso, alertas não podem avaliar *datapoints* e as telas com gráficos na parede do escritório não terão nenhum valor.
Sev1	Indisponibilidade total ou parcial pode impactar significativamente o negócio da empresa. Exemplo: Serviços cujas funcionalidades representam os aspectos mais importantes do produto. Geralmente categorizados como *tier* 1.
Sev2	Sistema severamente afetado, porém ainda operacional. Exemplo: Serviços cujas funcionalidades não estão completamente indisponíveis. Podem ser associados com serviços de *tier* 1 e 2.
Sev3	Geralmente atribuída a violações de SLO não tão severas. Exemplo: Um serviço cujo SLO possui três noves e que apresenta uma taxa de sucesso de 99% por um tempo suficiente impactando bastante usuários.
Sev4	Esta severidade está geralmente associada com situações em que times desejam conduzir uma investigação para tratar de um problema de baixa prioridade, por isso utilizam o processo de gerenciamento de incidente e ferramentas para iniciar uma ação coordenada. Também é um

Severidade	Sev0
	método para centralizar as comunicações e compartilhar qualquer progresso do esforço comum.

Incidentes de baixa severidade podem também ser escalados para severidades mais altas se uma mitigação não for encontrada em um tempo razoável ou se o problema se propagar e impactar outros serviços (como discutido na seção "Alertas e níveis de prioridade" do capítulo 10), então um incidente de severidade 2, por exemplo, pode, eventualmente, ser escalado para severidade 1 se usuários continuarem experimentando problemas por um longo período.

Declarar incidentes como do tipo `sev0` ou `sev1` geralmente é feito para eventos que devem ser propagados para toda a organização por questões de transparência, para que todos fiquem cientes do impacto e possam acompanhar a evolução do caso em direção à resolução.

12.3 Diagnóstico

Uma vez que o incidente é declarado, é essencial que seja resolvido rápido, por isso, o(a) SRE deve conduzir um diagnóstico e mergulhar em qualquer telemetria disponível para tentar encontrar a causa do problema. A resolução de problemas (do inglês, *troubleshooting*) é uma tarefa que pode seguir muitos caminhos dos quais devem ser tomados de acordo com o contexto do problema e as descobertas feitas no decorrer da investigação. Aqui vão algumas diretivas para auxiliar no processo de análise de um problema:

- A funcionalidade que está sendo diagnosticada já funcionou antes?
- Quando o problema ocorreu?

- Alguma coisa mudou recentemente?
- Alguma métrica, *trace* ou *log* apresenta alguma alteração suspeita logo quando o problema se manifestou?
- Existem circunstâncias específicas associadas com o problema?
- Outros componentes do sistema apresentaram algum comportamento que poderia estar relacionado com o problema?
- O problema pode ser reproduzido em outros ambientes?
- Isso poderia estar relacionado com permissões? Autenticação ou autorização?
- Alguma dependência na camada de infraestrutura foi atualizada recentemente (hardware, S.O. etc.)?
- Os recursos computacionais disponíveis para o serviço são suficientes?
- Você tentou reiniciar o serviço?

Essas perguntas servem apenas como exemplos de como raciocinar e reunir mais detalhes sobre um problema. Verificar se algo mudou recentemente é um ótimo método para analisar problemas, pois eventos como entregas de mudanças de código (*deployments*) ou alterações de configuração geralmente revelam-se como amostras promissoras de correlação e causalidade.

Tal diagnóstico pode precisar de um conhecimento mais aprofundado sobre os serviços envolvidos no problema, portanto é comum invocar especialistas, aqueles(as) responsáveis pelo sistema cuja funcionalidade em degradação causou o incidente, para que a colaboração facilite uma investigação mais efetiva. Se responsáveis pelo serviço em posição de engenheiro(as) de plantão primário não conseguirem progredir com a investigação em um tempo hábil, o(a) SRE deve sugerir o envolvimento do ponto de contato *secundário* do calendário de plantão ou até mesmo outros especialistas de outros times que possam ajudar a discutir ideias e progredir com a investigação.

12.4 Tratamento

Com a causa do problema identificada, ou mesmo que não seja totalmente compreendida, mas que um caminho para a resolução seja revelado, é hora de aplicar uma mudança para "curar" o sistema e restaurar a experiência dos usuários. A orientação mais importante que Comandantes de Incidentes devem manter em mente durante o gerenciamento de um incidente é: "Mitigação primeiro, soluções elegantes *DEPOIS*". Em outras palavras, se alguma configuração puder ser ajustada sem a necessidade do *deployment* de um código novo ou mesmo que alguma mudança de código simples seja uma opção, tais soluções rápidas devem tomar precedência sobre qualquer impulso de desenvolver uma nova função ou iniciar alguma jornada de engenharia, que pode consumir um tempo substancial, enquanto os clientes estão frustrados e a organização sofre impactos em sua reputação e suas finanças.

Durante o estágio de tratamento, Comandantes de Incidentes devem coordenar a sequência das ações que serão executadas para que elas não entrem em conflito entre si. Por exemplo: se um engenheiro A se oferecer para aplicar uma mudança de configuração de infraestrutura e outro engenheiro B quiser executar um *script* para alterar informações no banco de dados, em um cenário em que ambas as ações pareçam promissoras para restaurar o sistema, o(a) Comandante do Incidente deve decidir qual delas deve ser executada primeiro e acompanhar a telemetria para verificar seu resultado. Várias ações realizadas ao mesmo tempo, mesmo com o intuito válido de encontrar uma mitigação, podem criar mais problemas se forem executadas de forma caótica e podem gerar ruídos de comunicação e observação.

12.5 Revisão

O aprendizado adquirido através de incidentes é imperativo para a melhoria da confiabilidade de serviços, por isso SREs evangelizam e coordenam a reunião de "autópsia" do incidente, mais conhecida como *Post Mortem*. O formato de tal reunião pode variar de acordo com a organização, mas geralmente apresenta-se uma breve descrição do que aconteceu, os eventos que culminaram no problema discutido, sua detecção, a análise do impacto, a linha do tempo de todas as ações tomadas no decorrer do episódio, a definição de tarefas pós-incidente e aprendizados.

Ao revisitar os eventos que originaram um incidente, assim como os passos tomados durante os estágios de triagem, diagnóstico e tratamento, todas as entidades envolvidas discutem os detalhes do ocorrido e podem refletir sobre a telemetria disponível, o serviço; sua infraestrutura; e dependências; e cada uma das ações implementadas, revisando tudo de forma crítica sem a urgência de um incidente ativo. Desta forma, definem tarefas que devem ser delegadas de acordo com o escopo dos times envolvidos para melhorar a confiabilidade dos serviços envolvidos no incidente e evitar que problemas similares ocorram novamente. Para citar alguns exemplos comuns: implementação de testes adicionais, atualização de *runbooks*, criação de novos alertas ou revisões de arquitetura.

No final de uma reunião de *Post Mortem*, assim como em outras metodologias de retrospectivas, é uma prática comum que participantes ajudem a responder às seguintes perguntas:

- O que foi que deu certo? Se alguma ação ajudou na resolução do incidente, ela deve ser celebrada para melhorar a moral do time e replicada em outros casos para atingir os mesmos resultados positivos.
- O que foi que deu errado? Este é o momento em que se discute as falhas técnicas ou atritos em processos organizacionais utilizando experiência empírica de incidentes, aprendendo com os erros e elaborando propostas para melhoria contínua.
- O que aprendemos com esse incidente? Novos detalhes são revelados sobre os componentes do sistema e adquire-se novo

conhecimento sobre o comportamento de serviços e suas configurações diante de cenários incomuns. Também se aprende como os times interagem entre si durante tais cenários, o que revela oportunidades para estabelecer novos procedimentos para acelerar o processo de triagem ou mitigação.

É importante que as reuniões de *Post Mortem* sejam conduzidas com as seguintes orientações:

- **Sem atribuição de culpa:** geralmente expressada em inglês como *blameless*, ou seja, ninguém deve apontar dedos criando uma cultura de medo, engenheiros(as) devem rever o caso juntos(as) sem represálias, pois ninguém deve temer uma comunicação franca e honesta sobre qualquer erro humano que contribuiu para o surgimento do incidente, ninguém deve se esconder atrás de seu ego. O foco da reunião deve ser na melhoria dos aspectos tecnológicos do sistema e seus processos, se alguém cometeu um erro fazendo algo que não devia, devem ser discutidos mecanismos para proteger os componentes envolvidos e prevenir recorrências do mesmo problema.

- **Transparente:** todos(as) são bem-vindos(as) na reunião de *Post Mortem* e todos os detalhes documentados são acessíveis para qualquer membro da organização. Dessa forma não se criam silos de conhecimento, outras pessoas que fazem parte da engenharia podem se beneficiar do aprendizado obtido durante a revisão do incidente e os dados de uma página *wiki*, assim como o texto inserido em um sistema de mensageria, são indexados para que possam ser coletados por sistemas de busca em consultas futuras, ajudando outros a resolverem problemas similares que podem surgir no futuro.

- **Colaborativas:** cada participante pode oferecer uma perspectiva diferente ao analisar os eventos de um incidente, contribuindo com detalhes adicionais utilizando seu conhecimento técnico ou organizacional, assim como ideias e

sugestões que podem levar a organização ao próximo nível de excelência operacional.

Essas orientações podem ser brevemente revisitadas no início das reuniões para educar novos(as) engenheiros(as) e reiterar os parâmetros da discussão para veteranos.

Mesmo que alguma investigação exaustiva não apresente detalhes satisfatórios sobre a causa raiz de algum problema, os times envolvidos devem colaborar com melhorias direcionadas a ferramentas, telemetria ou processos para facilitar a detecção, prevenção ou mitigação de futuros incidentes. A revisão de incidentes deve promover evolução operacional, então qualquer que seja a tarefa definida durante tais discussões, ela deve beneficiar a próxima iteração do ciclo de vida de incidentes e trazer novos aprendizados para a organização. Dessa forma resta apenas o acompanhamento dessas tarefas para prosseguir com a conclusão do incidente.

12.6 Conclusão

Neste último estágio, Comandantes de Incidentes precisam apenas se certificar que todas as tarefas pós-incidente foram completadas e, finalmente, marcar o incidente como concluído.

Times de Engenharia de Confiabilidade de Sites devem conduzir uma revisão geral de incidentes periodicamente e analisar algumas métricas como:

- MTBF: Tempo Médio Entre Falhas (*Mean Time Between Failures*)
- MTTR: Tempo Médio de Recuperação (*Mean Time To Recover*)
- MTTD: Tempo Médio de Detecção (*Mean Time To Detect*)

Essas métricas podem ser fornecidas por ferramentas de gerenciamento de incidente e apresentar relatórios granulares para filtrar os resultados por serviços, times, severidade, categorias etc. O time deve discutir os resultados periodicamente e definir estratégias relacionadas a melhorias de confiabilidade de serviços, alocar mais recursos para ajudar times específicos ou desenvolver novas funcionalidades para ferramentas que podem facilitar o gerenciamento de incidentes.

SREs podem também reunir "histórias de guerra" (ou *war stories*) e compartilhar conhecimento adquirido em incidentes com outros times de engenharia. Isso é bastante útil em casos que recomendações comuns que podem ser aplicadas por outros times, para beneficiar e trazer resiliência para outros serviços. Tais histórias também podem ser compartilhadas em *blogs* de engenharia públicos, omitindo detalhes sensíveis e informações internas da organização, é claro, ou até mesmo através de palestras e conferências, para ajudar outros entusiastas de SRE e outros membros da comunidade que também passam seu aprendizado adiante.

Capítulo 13
Outras práticas para melhorias de confiabilidade

Neste capítulo, vamos explorar algumas práticas da indústria que podem ser adotadas para promover resiliência e melhor detectabilidade de incidentes. Cada organização possui suas próprias diretrizes para o desenvolvimento de seus serviços, a arquitetura de seus sistemas, assim como seus processos internos e a maneira como os times colaboram entre si, porém alguns padrões operacionais e procedimentos comuns são adotados devido ao sucesso demonstrado por grandes empresas que os utilizam e são famosos por sua escalabilidade, alta disponibilidade e a rápida recuperação de suas funcionalidades quando enfrentam incidentes.

13.1 Orientações básicas

Antes de discutirmos práticas mais elaboradas, temos que mencionar algumas orientações básicas que devem ser conhecidas por todos os membros da engenharia. Esta lista pode ser utilizada como um ponto de partida para que SREs possam documentar e evangelizar acordos internos que ajudam a prevenir ou reagir rapidamente a incidentes.

- Evite o "ponto único de falha" ou *Single Point of Failure* (SPOF), lembre-se de considerar redundância.
- Não inicie o *deployment* de uma mudança em Produção e dê as costas. Fique de olho na telemetria e certifique-se de que o novo código não está causando algum problema.
- Comunique qualquer evento suspeito e mantenha SREs cientes de qualquer acontecimento que pode impactar funcionalidades críticas e frustrar usuários.

- Sempre tenha um plano para reagir caso alguma mudança cause algum problema. Também é recomendado realizar uma revisão por pares (*peer review*) para que outros(as) engenheiros(as) possam avaliar a mudança, fornecer alguma crítica sobre o que está prestes a ser aplicado, avaliar o impacto e refinar o plano para reverter a ação se isso for uma opção válida.
- Mantenha os *runbooks* atualizados.
- Aprenda com os sucessos e falhas do sistema. Sempre agende um *Post Mortem* para incidentes de severidade 1.

Isso está longe de ser uma lista exaustiva, são apenas algumas orientações fundamentais que devem ser adotadas por times que desenvolvem e operam serviços em produção. Alguns dos pontos mencionados acima podem ser correlacionados com metodologias mais formais como a Biblioteca de Infraestrutura de Tecnologia da Informação (do inglês *Information Technology Infrastructure Library*, ou apenas ITIL) que possui diferentes tipos de gerenciamentos e princípios, incluindo a prática de criar Procedimentos Operativos Padronizados (*Standard Operating Procedure*) documentando os passos que devem ser seguidos para determinadas operações (para aprender mais sobre ITIL, visite https://www.ibm.com/topics/it-infrastructure-library).

Enfim, a lista deve ser expandida de acordo com a experiência adquirida ao fornecer suporte aos serviços da organização, lidando com suas operações rotineiras e eventuais incidentes.

13.2 Monitoramento Sintético

O Monitoramento Sintético (*Synthetic Monitoring*) é o ato de automatizar interações com algum sistema para monitorar o comportamento de suas funcionalidades, produzindo transações artificiais para avaliar suas funções e emitir resultados para ajudar

operadores(as) com a observação de serviços. Vamos considerar, por exemplo, um serviço X que recebe requisições de uma loja virtual e opera como uma API HTTP oferecendo múltiplas funções baseadas no verbo da requisição e no caminho de URL:

- `GET /produtos` : retorna uma lista de produtos.
- `GET /produtos/{id}` : retorna um produto específico baseado em um identificador único (*id*).
- `POST /busca?texto={termo_de_pesquisa}` : realiza uma busca no catálogo de produtos baseado no termo de pesquisa.
- `PUT /carrinho` : envia os dados de um produto no corpo da requisição para que ele seja adicionado ao carrinho de compras.
- `POST /compra` : finaliza a compra. Envia dados de pagamento para serviço de terceiros para iniciar transação, altera estoque, emite recibo etc.

Uma ferramenta de monitoramento sintético pode ser implementada para enviar requisições para cada uma dessas funcionalidades continuamente a cada minuto, confirmando que cada função está respondendo da forma esperada. Com estas requisições produzidas de forma contínua, o serviço é monitorado proativamente para que problemas sejam capturados antes mesmo que usuários reais tentem instrumentar a API do serviço X. Ao executar tais requisições em uma certa frequência, agindo como uma sonda (do inglês, *probe*), pode-se calcular a disponibilidade do serviço com base no período de observação e na contagem de respostas válidas em comparação com erros ou respostas lentas (minutos bons vs. minutos ruins) determinadas de acordo com algum limiar de tempo. Isso é bastante útil para produzir relatórios e medir a performance de serviços em relação aos seus respectivos acordos (SLAs ou SLOs).

Algumas ressalvas devem ser consideradas ao monitorar funções associadas com operações de escrita (*write*). Funções que criam novas entradas na camada de persistência não podem ser instrumentadas continuamente em ambientes de produção (exemplo: `POST /compra`), pois adicionam novos registros baseados em dados falsos criados por transações artificiais. Tais

funcionalidades tendem a ser excluídas do monitoramento sintético já que podem poluir o ambiente, afetando outros casos de uso e criando interações problemáticas com sistemas de terceiros (exemplo: tentar efetuar o pagamento de um produto ou alterar estoque). Alguns desses casos podem ser contornados com configurações que orquestrem uma sequência, em que alguma operação é executada e imediatamente revertida em cada ciclo de monitoramento, tornando a operação idempotente (que pode ser aplicada mais do que uma vez sem que haja alteração do resultado), para que iterações subsequentes não causem conflitos devido à existência de dados duplicados (ou à exaustão de um item no estoque, por exemplo). Outra prática comum é a utilização de um cabeçalho (*header*) HTTP para sinalizar os serviços que processam a requisição e suprimir a execução de algumas funções, o que impacta a "cobertura" do monitoramento, mas permite que algumas funcionalidades sejam instrumentadas e verificadas de acordo com as jornadas de usuário mais críticas.

Funções que executam apenas operações de leitura (*read only*), como `GET /produtos`, `GET /produtos/{id}` ou `POST /busca?texto={termo_de_pesquisa}` são geralmente mais simples em termos de configuração de monitoramento sintético, porém também possuem suas peculiaridades. Para citar algumas: em cenários em que a requisição está consultando dados (exemplo: `GET /produtos/{id}`), o monitoramento sintético pode até ser configurado com múltiplas sondas distribuídas geograficamente para avaliar a latência entre diferentes regiões ou *data centers* e averiguar que *clusters* de *cache* estão funcionando como esperado. Essa técnica exige mais infraestrutura e complexidade de implementação, mas garante uma boa segurança psicológica de que as requisições não estão enfrentando problemas de conectividade ou performance entre o cliente e os servidores que abrigam a aplicação.

Outro problema comum está relacionado com a ausência de um item específico que foi configurado como o alvo de uma consulta contínua, pois neste cenário a sonda do monitoramento sintético

está enviando requisições procurando por um *ID* arbitrário e, caso o registro do banco de dados seja marcado como expirado ou até mesmo acidentalmente deletado, alertas vão disparar e notificar times de engenharia, criando uma sobrecarga de comunicação desnecessária. SREs devem colaborar com os responsáveis pelos serviços para que tais registros que residem no ambiente de produção sejam "protegidos" de alguma forma para evitar o esforço operacional da triagem e restauração do monitoramento sintético (ou considerar a orquestração de operações idempotentes, como mencionado anteriormente).

Funções que interagem com sistemas de terceiros devem ser analisadas com cuidado já que a organização não tem autonomia para reagir a eventuais episódios de indisponibilidade de outras APIs. Portanto, cada caso deve ser avaliado com cautela, considerando inclusive sondagens experimentais para revelar se o monitoramento sintético de funcionalidades é viável ou se existem oportunidades para estabelecer procedimentos para melhorar a comunicação com o time de engenharia responsável por este sistema de terceiros. Quando um serviço depende de alguma API de terceiros, SREs devem discutir alguns requisitos operacionais para facilitar a triagem de alertas e incidentes em potencial. Isso pode incluir os seguintes questionamentos:

- Qual é o SLA desta API de terceiros?
- Existe alguma página de status que apresenta gráficos de disponibilidade e um histórico de incidentes? Tal página permite que usuários possam receber notificações em caso de incidentes?
- Que tipos de métodos de comunicação podem ser utilizados para notificar a engenharia deste time de engenharia externo em caso de incidentes? *Tickets* de suporte? Emails? Slack? Se isso envolve algum serviço de alta prioridade (*tier 1*), existem oportunidades para coordenar uma resposta mais imediata?
- Se algum problema é causado por falta de compatibilidade entre as APIs, esta organização externa pode facilitar a utilização de

algum ambiente do tipo "caixa de areia" (*sandbox*) para que mudanças possam ser verificadas em ambientes de Pré-produção antes que a versão que se encontra no ambiente de Produção seja alterada?

Como podemos ver, o monitoramento sintético possui alguns obstáculos, mas oferece o benefício de sondagem contínua e monitoramento proativo que pode ser implementado para alinhar-se a SLOs. Essa técnica definitivamente *não* substitui o monitoramento de tráfego orgânico, pois não pode observar efetivamente os resultados sob a perspectiva de usuários reais.

Para elaborar mais explicitamente, vamos supor que um serviço recebe `1000` requisições em 1 minuto e 2 delas retornam erros associados a uma função (um caminho de URL específico). Isso significa que a taxa de sucesso do serviço naquele minuto foi de `100 * (1 - (2 / 1000)) = 99.8`, portanto, se o serviço tem um SLO de três noves (`99.9%`), esse minuto representa uma violação de seu SLO que pode ser determinada facilmente com as métricas produzidas pelo tráfego de usuários reais. Já o monitoramento sintético tem uma baixa probabilidade de capturar erros em tais situações, já que sua sonda executa apenas uma única requisição naquele minuto.

O monitoramento realizado por transações artificais executadas continuamente servem o propósito de verificar proativamente se as funcionalidades de um serviço estão retornando as respostas esperadas e podem realizar asserções mais complexas do que uma simples taxa de sucesso baseada em monitoramento orgânico, como avaliar o conteúdo de uma resposta HTTP, por exemplo. Portanto, esses dois tipos de monitoramento não são mutualmente excludentes, um deve suplementar o outro para fins de relatórios e alertas de consumo de orçamento de erros.

13.3 Simulações de incidentes

As simulações de incidentes são atividades associadas com a Engenharia do Caos (*Chaos Engineering*), utilizadas para ensaiar uma situação de emergência e aprender sobre o comportamento de serviços em situações adversas.

A disciplina da Engenharia do Caos compreende os seguintes passos:

1. Definir os parâmetros para identificar um serviço saudável: mapear o estado de um serviço de acordo com a observação de sua telemetria em condições normais.
2. Criar uma ou mais hipóteses: conjecturar sobre as possíveis reações negativas que podem ser apresentadas pelo serviço. Por exemplo, ele pode sofrer uma interrupção abrupta em seu processamento, algum aumento de consumo de recursos computacionais (CPU, memória, disco etc.) ou impacto de performance (alta latência). Em outras palavras, determinar quais alertas devem ser disparados durante o exercício, se tais alertas existirem.
3. Introduzir injeção de falhas: criar cenários problemáticos causados pela exaustão de recursos ou produzindo erros em pontos estratégicos do sistema deliberadamente. Isso pode ser realizado através de ajustes na configuração da plataforma de *runtime* ou com a degradação proposital de dependências como outros serviços e componentes de infraestrutura, idealmente, através de automação.
4. Validar ou invalidar as hipóteses definidas anteriormente: coletar resultados dos exercícios para confirmar se os resultados esperados foram obtidos, se houve alguma aproximação ou se o comportamento apresentado desviou completamente do esperado.

Entretanto, sob a ótica de processos, além da validação de hipóteses relacionadas à tecnologia, essa prática também ajuda a criar uma "memória muscular" em relação a como se deve agir durante incidentes, como em procedimentos de simulações de incêndio. Da mesma forma, esses exercícios devem ser agendados

periodicamente para se certificar de que as pessoas saibam como proceder na hora de evacuar um prédio em chamas, evitando uma situação caótica em que o problema se agrava devido ao pânico e à ausência de processos.

No âmbito da engenharia de software, o "incêndio" é representado por um incidente, ou seja, uma funcionalidade severamente impactada que impede que usuários possam interagir com o sistema de forma satisfatória, como um alto volume de erros produzidos por múltiplos serviços, problemas de performance associados com funções críticas ou uma falha sistêmica. Basicamente, qualquer evento que cause um impacto negativo para o negócio da organização ou sua reputação, incluindo episódios de violações de segurança onde o acesso ao sistema ou os dados pessoais identificáveis (do inglês, *Personally Identifiable Information* ou apenas PII) podem estar comprometidos.

Comandantes de Incidente devem coordenar as ações das pessoas envolvidas, sempre priorizando decisões que levem à mitigação para fazer com que serviços voltem a operar dentro de seus respectivos SLOs, porém, ao planejar simulações, artefatos como alertas, as seções do *runbook*, sistemas de escalação e *pipelines* de *deployment* já podem ser revisados antes mesmo de o exercício acontecer, o que ajuda a preparar melhor engenheiros(as) de plantão (*on-call*) que devem interagir com esses recursos para responder a eventos problemáticos.

O propósito dessas simulações, como explicado no livro *Engenharia do Caos* (Rosenthal e Jones, 2020), *"não é causar desorganização e desconforto, pelo contrário, é fornecer uma situação controlada na qual humanos podem aprender a navegar o caos em direção ao resultado esperado, que neste exemplo é a remediação rápida de um incidente duradouro"*.

Então, de certa forma, uma simulação de incidente é praticamente um teste de integração aplicado ao processo de gerenciamento de incidente, onde a sequência de passos produz resultados que são

verificados com suas respectivas asserções. Portanto, essa prática ajuda times de engenharia a:

- Melhorar seus *runbooks*.
- Confirmar que sondas de monitoramento sintético, métricas, *logs*, *traces* e alertas estão funcionando como esperado.
- Testar *backups*.
- Averiguar o funcionamento de sistemas de escalação e o envio de *pages* para os pontos de contato primários e secundários.
- Analisar a eficiência de processos de *release* para *bug fixes*, alterações em interruptores de funcionalidades ou mudanças em configurações de infraestrutura.

Imagine o quão frustrante seria receber um *page* às 3 da manhã, iniciar um incidente de alta severidade porque o *runbook* do serviço impactado não contém nenhuma informação sobre um alerta específico e, logo depois de consumir muito tempo e resolver o incidente aplicando um esforço significativo envolvendo múltiplas pessoas, ao discutir o episódio durante uma reunião de *Post Mortem*, é revelado que existia um passo trivial que poderia recuperar o sistema facilmente, mas que nunca foi documentado. Ou até mesmo, ao tentar escalar o problema e entrar em contato com desenvolvedores(as) de plantão, descobre-se que esqueceram de instalar ou configurar o aplicativo de recebimento de *pages* em seus dispositivos móveis, causando uma sobrecarga de comunicação já que o incidente precisa ser escalado para a camada de gerência e outros membros da engenharia.

A disciplina da Engenharia do Caos revela tantas oportunidades para amadurecimento de processos e melhoria na resiliência de serviços que algumas ferramentas ficaram bastante famosas na indústria: a primeira é o *Chaos Monkey*, desenvolvida pelo Netflix, que destrói máquinas virtuais (nós de um cluster) de forma randômica para incentivar times de engenharia a alocar recursos para tornar seus serviços mais robustos. A segunda ferramenta que deve ser mencionada é o "Serviço de Injeção de Falha" (*Fault Injection Service* ou FIS) do Amazon Web Services (AWS), que permite que

operadores(as) definam que tipo de problema deve se manifestar em um determinado recurso da infraestrutura, por exemplo, remover instâncias *EC2*, criar gargalos na rede ou adicionar latência em serviços específicos. Essas injeções de falhas revelam oportunidades para rever certos aspectos da engenharia de serviços como: evitar o "ponto único de falha" (SPoF), definir estratégias de respostas padrões sensíveis em caso de falhas (*fallback*) e a remoção de dependências, mantendo-se um número mínimo de interações com outros serviços essenciais. Essas mudanças são geralmente implementadas com o aprendizado adquirido depois de enfrentar múltiplos incidentes, porém, com ferramentas dessa natureza, estas epifanias podem ser alcançadas de forma mais rápida, proativamente e em situações controladas pela própria engenharia.

Por isso, grandes empresas como o Google realizam exercícios periódicos de simulações de desastres, através de programas como o "Teste de Desastre e Recuperação" (DiRT), como mencionado no capítulo 4. Para organizações menores, a prática de simulações de incidentes pode ser iniciada com procedimentos mais simples executados manualmente, ajustes à configuração de infraestrutura para diminuir a capacidade computacional de um serviço e ver como ele se comporta durante um teste de carga, por exemplo. Ou até mesmo um exercício que envolve apenas um incidente falso e o envio de um *page* para engenheiros(as) de plantão, para se certificar de que o contato pode ser estabelecido em caso de alguma emergência real. Comece com simulações simples e refine o processo e a utilização de ferramentas à medida que os exercícios são conduzidos, pois não se deve postergar a chance de aprender mais sobre o comportamento de serviços nessas situações anormais. Como um bônus, ao convidar os times responsáveis para tais simulações, esta é uma grande oportunidade de conhecer outros membros da engenharia.

13.4 Interruptores de funcionalidades

Interruptores de funcionalidades (*Feature Toggles* ou *Feature flags*) permitem que operadores(as) alterem o comportamento de um serviço instantaneamente através de uma configuração remota. Seja através de alguma condição binária que pode ser usada para ativar ou desativar algum comportamento como `{ "exibir_aviso_de manutencao": [ true | false ] }`, ou uma configuração mais complexa com um ou mais parâmetros e valores que podem fornecer instruções mais granulares para o serviço (exemplo: `{ "banner_de_promocao": "https://loja-xyz/img/descontos_de_verao.jpg", "pontos_de_fidelidade": 20 }`). Essas configurações devem ser obtidas pelas aplicações em tempo de execução (*runtime*) e são utilizadas de forma estratégica para fins operacionais e comerciais.

Uma organização pode tirar proveito de interruptores de funcionalidades para lançar funcionalidades experimentais em seu serviço (website, aplicativo móvel, API etc.), acompanhar a telemetria produzida pelos componentes do sistema ou relatórios de mercado, e desligar a "chave" de tal experimento a qualquer momento caso não esteja obtendo a resposta desejada, seja devido a algum impacto de performance da tecnologia envolvida ou algum resultado indesejado sob a perspectiva de mercado.

Essa técnica também compreende a definição de segmentos (*segments*) para que tais interruptores sejam aplicados a grupos específicos da base de usuários. Alguns exemplos de segmentos incluem a ativação de novos comportamentos para um percentual de usuários, localização geográfica ou o tipo de usuário, caso exista um modelo de assinatura do serviço com mais privilégios, como um acesso *premium*, por exemplo. A segmentação de usuários e observação de resultados é bastante útil para fins de avaliações de *marketing*, conhecidas como "Testes A/B", pois revela que tipos de ofertas ou campanhas tornam-se bem-sucedidas entre grupos específicos de usuários.

Além de suas vantagens em termos de *marketing*, interruptores de funcionalidades são particularmente úteis para SREs já que oferecem um mecanismo muito eficiente para resolução rápida de problemas. Como mencionado no capítulo 10, na seção "Documentando passos acionáveis para alertas", se um *runbook* possuir instruções para instrumentar alguma *feature flag* caso algum alerta específico seja disparado, SREs podem prevenir ou mitigar um incidente rapidamente sem a necessidade da sequência de ações envolvidas no *deployment* de uma mudança de código ou de infraestrutura.

Através da configuração de segmentos, uma ferramenta de gerenciamento de interruptores de funcionalidades também oferece a habilidade de ativação gradativa de novos comportamentos, que está alinhada com a estratégia de *deployment* do tipo Canário (como visto no capítulo 3), permitindo que operadores(as) disponibilizem uma nova funcionalidade para, por exemplo, apenas 1% dos usuários. Nesse cenário, uma pequena amostra de usuários utiliza uma nova função B, enquanto todos os outros continuam obtendo os resultados produzidos por uma antiga função A. Como a função original foi testada exaustivamente pela utilização prévia no ambiente de produção, assume-se que ela oferece uma funcionalidade mais estável e amadurecida, o que significa que qualquer impacto que venha a ser produzido pela nova função B será isolado apenas para 1% dos usuários.

Essa técnica permite um controle granular e estratégico que disponibiliza esta nova função para um grupo de usuários para fins de observar a performance do serviço, analisando a telemetria e SLIs para confirmar que não há nenhuma regressão em termos de estabilidade. Se a nova função apresenta resultados positivos, o segmento é reconfigurado progressivamente em uma sequência de `10%`, `25%` e `50%` de adoção, por exemplo, de acordo com os resultados e, se algum problema for observado, configuração é revertida para zero (`0%`) pois a experiência do usuário ou o estado do sistema precise ser recuperado o mais rápido possível.

É uma boa prática criar notificações para eventos relacionados a alterações de interruptores de funcionalidades, pois é muito comum procurar por mudanças recentes quando alertas são disparados. Assim como *deployments* de mudanças de código geralmente estão sempre na lista de suspeitos de correlação e causalidade, *feature_flags* também podem causar incidentes. Por isso, tais notificações e suas respectivas informações de auditoria ("quem modificou", "o que foi modificado" e "quando") com dados exatos de data e hora são extremamente importantes para "conectar os pontos" quando algum problema é reportado. Tais notificações associadas com interruptores de funcionalidades podem ser propagadas para canais de sistemas de mensageria ou ativar recursos de outras ferramentas. Por exemplo, a plataforma de observabilidade Grafana permite que "Anotações" (*annotations*) sejam adicionadas aos painéis onde as métricas de serviços são visualizadas, que é muito útil para fins de correlação de eventos.

Para criar uma anotação do Grafana você vai precisar de um código de conta de serviço (*service account token*) para interagir com a API. Verifique a documentação oficial sobre como obter o *token* (https://grafana.com/docs/grafana/latest/administration/service-accounts/migrate-api-keys/) e o armazene em uma variável em seu terminal (exemplo: `GRAFANA_SERVICE_ACCOUNT_TOKEN`). Agora que você possui a autorização para utilizar a API, os seguintes parâmetros devem ser preenchidos para criar uma anotação:

- `dashboardUID` : o identificador único (*UID*) do dashboard pode ser encontrado na sua URL.
- `panelId` : o identificador do painel também pode ser encontrado na URL quando for editado.
- `time` : a data/hora em que o evento ocorreu (em formato Era UNIX ou *Unix epoch*, que é a contagem de segundos desde o dia 1 de janeiro de 1970).
- `timeEnd` : a data/hora em que o evento terminou.
- `tags` : marcadores utilizados para categorizar a anotação (anotações podem ser consultadas em *dashboards* e aplicados a

múltiplos painéis).

- `text` : texto descrevendo o evento.

Aqui está um exemplo de como criar uma anotação de um evento com a duração de 1 minuto utilizando os parâmetros descritos acima:

```
ANNOTATION_START="$(date +%s)000"
ANNOTATION_END="$(( $(date +%s)000 + 60000 ))"

curl -vvv -H "Authorization: Bearer ${GRAFANA_SERVICE_ACCOUNT_TOKEN}" -H "Content-Type: application/json" -X POST "http://grafana-local.com.br/api/annotations" --data-raw '{"dashboardUID": "abc123456z","panelId": 17, "time": '$ANNOTATION_START', "isRegion":true,"timeEnd": '$ANNOTATION_END', "tags": ["feature_flag:exibir_aviso_de manutencao", "experimental"], "text":"interruptor do servicoX modificado devido a manutencao do banco de dados"}'

curl -s -H "Authorization: Bearer ${GRAFANA_API_KEY}" -H "Content-Type: application/json" -X POST "http://grafana-local.com.br/api/annotations" --data-raw '{"dashboardUID": '6f9tpnd7z',"panelId": 8, "time": '$ANNOTATION_START', "isRegion":true,"timeEnd": '$ANNOTATION_END', "tags": ["feature_flag: 'exibir_aviso_de manutencao'", "experimental"], "text":"interruptor do servicoX modificado devido a manutencao do banco de dados"}'
```

Esse comando vai gerar uma anotação no painel `8` do dashboard cujo UID é `6f9tpnd7z` , aqui está uma captura de tela ilustrando como anotações servem como um recurso poderoso para correlação e causalidade.

Figura 13.1: Exemplo de anotação do Grafana

Apesar dos benefícios de interruptores de funcionalidades, devemos lembrar que um grande número de comportamentos diferentes implica em uma sobrecarga de escopo de testes. Se um serviço possui muitas funcionalidades que podem operar de alguma maneira diferente dinamicamente, é exigido esforço adicional para que se alcance uma "cobertura" satisfatória de testes unitários, testes de integração e até mesmo testes de carga. Um serviço que possui muitas funções controladas por diferentes interruptores de funcionalidades apresenta um alto número de permutações que devem ser testadas, o que introduz mais complexidade e consome mais tempo das *pipelines* que compreendem o ciclo de vida de desenvolvimento. Portanto, *feature flags* não devem ser consideradas em todas as circunstâncias. Essa técnica é definitivamente uma adição fundamental à bagagem de conhecimento de SREs, porém sua aplicação deve ser analisada cuidadosamente, geralmente sendo mais considerada para funcionalidades de alta prioridade alinhadas com o produto e as jornadas críticas de usuários.

13.5 Redirecionamento de tráfego

Esta prática está geralmente associada com serviços que possuem uma arquitetura de múltiplas regiões (*multi-region architecture*), já que assume que existem diferentes *clusters* da aplicação em diferentes *data centers*. Caso algum deles apresente algum problema, operadores(as) têm a habilidade de redirecionar o tráfego (termo conhecido em inglês como *site failover*) para que os usuários possam interagir com instâncias saudáveis em uma cópia estável do ambiente de produção.

Para ilustrar, imagine que a pilha de tecnologia de um determinado serviço de *back-end*, incluindo todas suas dependências e os respectivos bancos de dados, existem nas regiões `us-west-1` e `us-east-1`, ou seja, em *data centers* na costa oeste e na costa leste dos Estados Unidos. Os dois clusters servem um número considerável de usuários que acessam o website de uma loja virtual, enviando milhares de requisições para o serviço de *back-end* em ambas as regiões. O time responsável pelo serviço costuma monitorar a taxa de sucesso do serviço através de um monitor na parede do escritório (que também é observado pelo time de SRE) e, em uma tarde fatídica, um dos engenheiros percebe a ocorrência de erros no cluster da região `us-east-1` que vai além do limiar do SLI e leva o serviço a violar seu SLO (99.9% neste caso).

Isso significa que existem usuários frustrados na região leste:

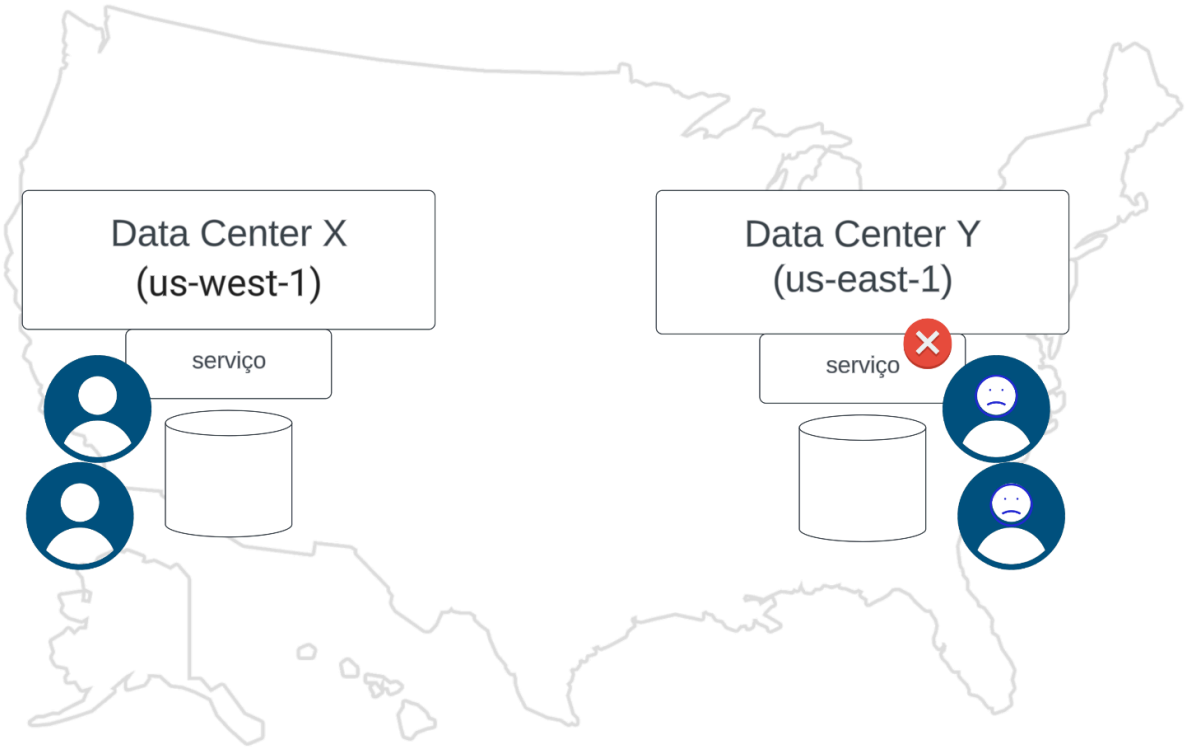

Figura 13.2: Erros na região leste

Se esta fosse a única região em que o serviço está presente, SREs precisariam seguir uma sequência de passos para lidar com este problema, por exemplo: iniciar uma triagem, verificar o *runbook*, determinar impacto, anunciar um incidente, analisar a telemetria do serviço em *dashboards*, tentar encontrar correlação e causalidade em seus painéis, verificar se o problema está relacionado a alguma saturação de recursos ou algum *bug* em uma mudança de código aplicada através de algum *deployment* recente, enviar um *page* para especialistas, aplicar alguma mudança de configuração de escalabilidade da infraestrutura ou iniciar a *release* de um *bug fix* para mitigar o incidente e, se tudo correr bem, resolver o problema e restaurar a experiência dos usuários.

Porém, como neste cenário temos a vantagem de operar em duas regiões, em vez de proceder com os passos citados anteriormente, o(a) SRE pode apenas verificar o estado do cluster na outra região, certificando-se de que o serviço ainda opera dentro de seu SLO na

região leste e redirecionar parte do tráfego da costa leste para o *data center* estável em `us-west-1`.

Vamos supor que as instâncias do serviço que apresentam erros na costa leste estão enfrentado problemas de performance associados com uma réplica de um banco de dados de cache. Para elaborar mais, digamos que esta réplica está processando muitas transações e está sob bastante pressão (saturação de recursos), portanto a SRE decide redirecionar `25%` do tráfego da região `us-east-1` para `us-west-1` e aliviar a carga de trabalho na região leste.

Ao consultar a ferramenta de observabilidade, a SRE determina que o incidente foi mitigado:

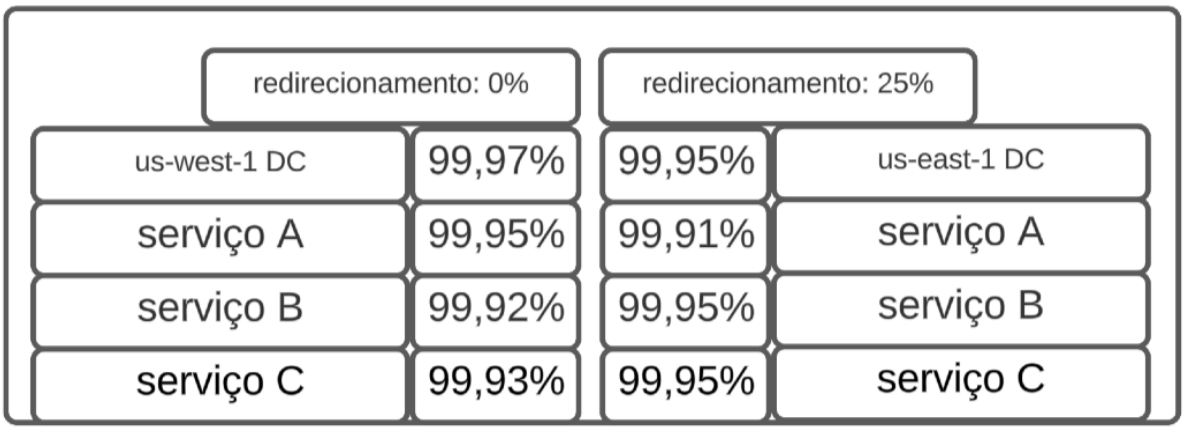

Figura 13.3: Exemplo de monitoramento de serviços em múltiplas regiões

Agora, a SRE pode continuar com os passos adicionais do ciclo de vida do gerenciamento de incidente, enviar um *page* para o time de plantão responsável pelo serviço para dar início a uma investigação mais detalhada sobre o que aconteceu com o sistema de cache da região oeste. Porém, enquanto essa investigação é conduzida, devido à técnica do redirecionamento de tráfego, a causa raiz do incidente pode ser analisada enquanto os usuários navegam pelo website e instrumentam suas funcionalidades sem problemas, permitindo que o sistema apresente um tempo de indisponibilidade mínimo.

Esse é apenas um exemplo para ilustrar a incrível vantagem de uma arquitetura de múltiplas regiões. Com esta configuração, surgem possibilidades interessantes como:

- Redirecionamento do tráfego de um serviço entre *data centers* para permitir alguma manutenção em uma região específica (exemplo: atualização de bancos de dados ou realizar experimentos com outros componentes de infraestrutura), seguido do "rebalanceamento" do tráfego (reajustar o percentual de redirecionamento para `0%` em ambas as regiões).
- Realizar o *deployment* para uma única região e executar uma *release* progressiva (canário) através do redirecionamento de tráfego, assim como Testes A/B em que algumas funcionalidades podem ser disponibilizadas apenas em uma região.
- Exercícios de redirecionamento completo de sites (*full site failover*), movendo o tráfego de *TODOS* os serviços para se certificar que um único *data center* pode acomodar todas as requisições para fins de continuidade em casos de desastre (caso uma região inteira torne-se indisponível).
- Lidar com incrementos de tráfego sazonais (que ocorrem em determinada época do ano), por exemplo: vamos dizer que os servidores da costa leste ficam mais ocupados durante a Black Friday, então ativa-se o redirecionamento de `25%` do tráfego para a costa oeste).

O redirecionamento de tráfego é uma técnica poderosa para SREs e deve ser utilizada com cautela. Se algum percentual de tráfego muito alto for colocado em um comando de redirecionamento, dependendo do volume de requisições, o aumento repentino de transações e a utilização agressiva de recursos computacionais em um curto espaço de tempo pode causar graves problemas em um *data center*. Por isso, operadores(as) deste tipo de ferramenta devem proceder com cautela e redirecionar gradativamente e em espaços de tempo razoáveis, de acordo com os padrões de tráfego e a capacidade dos clusters de serviços e sua infraestrutura. Uma vez

que tais padrões forem identificados, SREs podem desenvolver mitigações automáticas baseadas em simples algoritmos, como:

```
Se a taxa de sucesso do serviço X ficar abaixo de 99,9% por > 3 mins em
us-east-1, redirecione 25% do tráfego para us-west-1.
```

O *design* e implementação de serviços que acomodam arquiteturas de múltiplas regiões nem sempre são uma tarefa fácil, devido a diferentes tipos de serviço e seus padrões de acesso aos dados. Portanto, essa técnica só pode ser aplicada com a colaboração intensa com desenvolvedores(as). Lembre-se também de que as métricas e outros recursos de telemetria, assim como as ferramentas de observabilidade, devem ser configuradas para suportar a utilização de múltiplas regiões.

13.6 Considerações finais

Agora que você conhece os conceitos da Engenharia de Confiabilidade de Sites e as responsabilidades das pessoas que colaboram entre si para implementar essas práticas, é hora de aplicar este conhecimento na organização na qual você trabalha. No decorrer destes capítulos, nós exploramos vários tópicos relacionados à Engenharia de Software e como eles estão relacionados com SRE, portanto, aqui estão algumas ideias que podem lhe ajudar a iniciar novos projetos nessa área:

- Identifique os processos e artefatos que compreendem o ciclo de vida do desenvolvimento de software, ambientes, tipos de *release*, assim como as pipelines de Integração Contínua e Entrega Contínua configuradas pelos times de desenvolvimento. Avalie quais os pontos que apresentam oportunidades de melhoria.
- Analise as arquiteturas adotadas pelos serviços mais críticos que compõem as funcionalidades com as quais os clientes interagem. Tente mapear as dependências de infraestrutura

mais comuns, principalmente as plataformas de *runtime*, e familiarize-se com seus mecanismos de configuração e interfaces operacionais para facilitar atividades de suporte e revisão de requisitos não funcionais.
- Formule um plano de monitoramento de serviços utilizando as estratégias e opções de telemetria discutidas neste livro, colaborando com desenvolvedores para que métricas, *traces* e logs sejam produzidos pelos serviços seguindo padrões que facilitem o provisionamento de *dashboards* e alertas.
- Elabore uma lista de requisitos para serviços que operam no ambiente de Produção. Ajude outros membros da engenharia com iniciativas de "encontrabilidade" (*Discoverability*) para facilitar esforços relacionados a suporte e operações, como *runbooks* e a elaboração de um catálogo de serviços.
- Apresente os benefícios da definição de SLIs e SLOs. Explique como tais acordos de nível de serviço facilitam a geração de relatórios de performance, orientando desenvolvedores(as), gerentes e outras entidades em camadas executivas, a tomarem decisões sobre a necessidade de alocação de recursos para aumentar a resiliencia de um ou mais serviços, incluindo as vantagens para engenheiros(as) de plantão em relação à detecção, triagem e medição do impacto de eventuais problemas encontrados no ambiente de produção.
- Torne-se o(a) pioneiro(a) do processo de gerenciamento de incidentes na sua organização. Evangelize boas práticas de aprendizado contínuo, organize reuniões de *Post Mortems* e simulações de incêndio para se certificar de que os engenheiros de plantão estão com os canais de comunicação devidamente configurados e que o processo seja conhecido por todas as pessoas que podem, eventualmente, estar envolvidas em algum incidente.
- Automatize tarefas repetitivas e desenvolva ferramentas para facilitar tarefas rotineiras.

Mesmo que sua empresa já possua alguns destes projetos em andamento, ou até mesmo bem-estabelecidos, a jornada de SRE

está sempre em evolução pois novas tecnologias, arquiteturas, padrões de acesso de dados e soluções de observabilidade estão surgindo a cada dia que passa. SREs devem se atualizar e explorar outras áreas para fornecer suporte e colaborar com especialistas, adaptando os processos e ferramentas existentes para fins de monitoramento e definição de acordos de nível de serviço.

Despedida

Chegamos ao fim desta jornada. Espero que o conteúdo deste livro sirva como inspiração para desbravar um novo mundo focado no ímpeto de observar múltiplos componentes de sistemas complexos e buscar a melhoria da resiliência de serviços. Que este material ajude você com novas aventuras caso esteja considerando alguma mudança de carreira ou que desperte ideias para projetos de engenharia de software para melhorar ainda mais a confiabilidade de sua organização. Obrigado pela leitura!

Capítulo 14
Bibliografia

BEYER, JONES, MURPHY, PETOFF. *Site Reliability Engineering:* How Google Runs Production Systems, O'Reilly Media, 2016. Disponível em: https://research.google/pubs/site-reliability-engineering-how-google-runs-production-systems/

BEYER, KAWAHARA, MURPHY, RENSIN, THORNE. *The Site Reliability Workbook:* Practical Ways to Implement SRE, O'Reilly Media, 2018. Disponível em: https://research.google/pubs/the-site-reliability-workbook/

CHALUP, HOGAN, LIMONCELLI. *The Practice of Cloud System Administration:* Designing and Operating Large Distributed Systems, Volume 2, Pearson Education, 2015.

FONG-JONES, MAJORS, MIRANDA. *Observability Engineering*. O'Reilly Media, 2022.

14.1 Recomendações

Livros

BLANK-EDELMAN. *Seeking SRE:* Conversations About Running Production Systems at Scale. O'Reilly Media, 2018.

JONES, ROSENTHAL. *Chaos Engineering:* System Resiliency in Practice. O'Reilly Media, 2020.

PARKER, YOUNG. *Learning OpenTelemetry:* Setting Up and Operating a Modern Observability System, O'Reilly Media, 2024.

STORLASKY, WOO. *97 Things Every SRE Should Know:* Collective Wisdom from the Experts. O'Reilly Media, 2020.

Outros rescursos

DEVOPSLINKS. (Newsletter). Disponível em: https://faun.dev/newsletter/devopslinks. Acesso em: 21 Dezembro. 2024.

EMBRACE RISK. (Newsletter). Disponível em: https://embracerisk.substack.com. Acesso em: 21 Dezembro. 2024.

GRAFANALABS BLOG ENGINEERING POSTS. (Blog). Disponível em: https://grafana.com/categories/engineering. Acesso

em: 21 Dezembro. 2024.

HIGH SCALABILITY. (Blog). Disponível em: https://highscalability.com. Acesso em: 21 Dezembro. 2024.

HIPSTERS PONTO TECH. (Podcast). Disponível em: https://www.hipsters.tech. Acesso em: 21 Dezembro. 2024.

KUBICAST. (Podcast). Disponível em: https://kubicast.buzzsprout.com/. Acesso em: 26 Dezembro. 2024.

NETFLIX TECH BLOG. (Blog). Disponível em: https://netflixtechblog.com. Acesso em: 21 Dezembro. 2024.

SOFTWARE ENGINEERING RADIO. (Podcast). Disponível em: https://se-radio.net. Acesso em: 21 Dezembro. 2024.

HARDCODE PODCAST. (Podcast). Disponível em: https://podcast.app/hardcode-podcast-p982131. Acesso em: 26 Dezembro. 2024.

SRE WEEKLY. (Newsletter). Disponível em:

https://sreweekly.com. Acesso em: 21 Dezembro. 2024.

TWITTER ENGINEERING. (Blog). Disponível em: https://blog.x.com/engineering/en_us/topics/infrastructure. Acesso em: 21 Dezembro. 2024.

WAR STORIES. (Podcast). Disponível em: https://warstories.criticalpoint.tv. Acesso em: 21 Dezembro. 2024.